LICENCE SCIENCES & TECHNOLOGIES
1re ANNÉE

UE INF201, INF231
ALGORITHMIQUE ET PROGRAMMATION FONCTIONNELLE
2019 / 2020

PROJET — LE JEU DU RUMMIKUB

Objectifs

Le but de ce projet est de programmer le jeu du *Rummikub*. L'objectif est de vous entraîner à manipuler les séquences, qu'elles soient modélisées par un type somme ad hoc ou bien par des listes. Ces structures de données récursives sont fondamentales en informatique en général, et en programmation fonctionnelle en particulier. Vous :

- définirez d'abord des fonctions afin de pouvoir manipuler des multi-ensembles ;
- ré-usinerez ensuite ces fonctions, notamment en utilisant l'ordre supérieur ;
- implémentererez un « gameplay » permettant de jouer au Rummikub.

La réalisation des deux premiers points ne nécessite aucune connaissance des règle du jeu du Rummikub.

Déroulement du projet

Ce projet se réalise en binôme pendant les séances de TP non supervisées (et en pratique libre). Il se décompose en deux phases à l'issue desquelles vous devrez effectuer une soutenance sur machine et devant votre enseignant de TD/TP, pendant la dernière séance de TP du semestre.

Les livrables de chacune des phases sont constitués par un seul fichier de code source OCAML commenté que vous devrez rendre par email (1 mail par binôme) à vos enseignants :

1) au plus tard le 29 mars à 23h59 : Q1 (partie 3) et Q2 (partie 4) ;
2) au plus tard la veille du jour des soutenances à 23h59 : les autres questions.

Les modalités de la soutenance et des rendus des livrables seront précisés avec vos enseignants.

Remarque Faire Q1 avant le 9 mars constitue un bon entraînement pour le partiel.

La version éléctronique de ce document est disponible (en couleur, et format A4) à l'adresse : `http://membres.imag.fr/puitg/Ens/201/`. Les corrections et mises à jour évenuelles seront publiées à cette adresse.

Table des matières

1 Introduction **3**

2 Multi-éléments **3**

3 Multi-ensembles **3**
3.1 Fonctions opérant sur les multi-ensembles 4
3.2 Implémentation des multi-ensembles sur un type somme 5

4 Réusinage **6**

5 Le Rummikub **6**
5.1 Règles du jeu 7
5.1.1 Règle 1 : constitution de la table de jeu 7
5.1.2 Règle 2 : distribution initiale 7
5.1.3 Règle 3 : tour de jeu 7
5.1.4 Règle 4 : combinaisons valides 8
5.1.5 Règle 5 : première pose 8
5.1.6 Règle 6 : arrêt du jeu 8
5.1.7 Règle 7 : calcul du score et joueur gagnant 8

6 Modélisation des données **8**
6.1 Les tuiles 8
6.2 Combinaisons, tables et poses 9
6.3 Mains et pioches 9
6.4 Les joueurs 10
6.5 État d'une partie 10
6.5.1 État initial 11
6.5.2 Accès aux informations d'un état 11

7 Mise en œuvre des règles **11**
7.1 Validité des combinaisons 12
7.2 Calcul de points 12

8 Gameplay **13**
8.1 Validité des coups 13
8.2 Ajout d'une tuile sur la table 13
8.3 Recherche de combinaisons dans une main 13
8.4 Changement d'état 14
8.5 Scenario de jeu 15

9 Pour aller plus loin **16**

Barème. Le projet est noté sur 100. Le barème est indicatif. La réalisation parfaite des sections 3 et 4 fera tendre la note assymptotiquement (et inférieurement) vers 50. Pour passer la barre de la moyenne, il faudra donc faire un effort de plus.

Il est possible d'avoir 100 sans faire les questions de la section 9, qui sont hors barème.

Utiliser d'autres fonctions que celles de cet énoncé n'est autorisé qu'à condition de les **spécifier avant implémentation** (profil, sémantique, exemples ou propriétés).

Sauf mention contraire, l'utilisation de fonctions de la librairie standard d'OCaml est **interdite**. En cas de doute, demandez à votre enseignant.

1 Introduction

Comme toujours en informatique, il est nécessaire de pouvoir représenter des groupements de valeurs. Les valeurs sont prises dans un *réservoir*, noté α. Par exemple, si $\alpha = \mathbb{N}$, on groupe des entiers naturels.

Il existe de multiples façons de regrouper des valeurs. Ce projet en examine une en particulier : les multi-ensembles.

Un $multi-ensemble_\alpha$ est un groupement non ordonné de valeurs prises dans α.

Contrairement aux ensembles, on autorise les valeurs à être répétées ; il est donc possible qu'il y ait plusieurs occurrences (exemplaires) de la même valeur. Par exemple :

$$ex\_ens \stackrel{def}{=} \{3, 4, 6, 3, 6, 6\}$$

est un $multi-ensemble_\mathbb{N}$ contenant deux occurrences de 3, une occurrence de 4 et trois occurrences de 6.

2 Multi-éléments

Les multi-ensembles autorisant de multiples occurrences d'une même valeur, nous introduisons la notion de multi-élément. Un *multi-élément* représente toutes les occurrences d'une même valeur appartenant à un multi-ensemble.

On peut modéliser un multi-élément par un couple formé d'un élément du réservoir α et d'un entier naturel appelé *multiplicité* :

$$multielement_\alpha \stackrel{def}{=} \alpha \times \mathbb{N}$$

Par exemple, si $\alpha = \{'a', \ldots, 'z'\}$, $('m', 3)$ est un multi-élément, de multiplicité 3. La multiplicité d'un élément indique le nombre d'occurrences de l'élément.

On implémentera ces notions en OCaml grace au type polymorphe :

```
type nat = int  (* ≥ 0 *)
type 'a multielement = 'a * nat (* 'a ≡ α : réservoir des valeurs *)
```

3 Multi-ensembles

25 pts

On peut maintenant formaliser la définition donnée en introduction :

$$multiensemble_\alpha \stackrel{def}{=} \{\emptyset\} \cup \{C((x, m), ens) \;/\; (x, m) \in multielement_\alpha,\ ens \in multiensemble_\alpha,\ \forall m' \in \mathbb{N}, (x, m') \notin ens\}$$

Dans cette définition :

- $\emptyset$ dénote le multi-ensemble vide ;
- (x, m) est un $multielement_\alpha$, x étant un élément du réservoir et m sa multiplicité ;
- $C((x, m),\ ens)$ dénote un multi-ensemble non vide comportant (au moins) le multi-élément (x, m).

Comme on le voit dans cette formalisation, les valeurs ne sont pas stricto sensu répétées, mais du fait de leur multiplicité, tout se passe comme si elles existaient en plusieurs exemplaires. Par exemple, le multi-ensemble $C(('m', 3),\ C(('u', 1),\ \emptyset))$ « contient » 3 occurrences de 'm' et 1 occurence de 'u'.

3.1 Fonctions opérant sur les multi-ensembles

Afin de pouvoir manipuler des multi-ensembles, nous spécifions les fonctions suivantes :

1. Cardinalité d'un multi-ensemble :

   Profil $cardinal : multiensemble_\alpha \to \mathbb{N}$

   Sémantique : *$cardinal(ens)$ est le nombre total d'occurences des éléments de ens.*

   Exemple : $cardinal(ex\_ens) = 6$

2. Nombre d'occurrences d'un élément :

   Profil $nbocc : \alpha \to multiensemble_\alpha \to \mathbb{N}$

   Sémantique : *$(nbocc\ elt\ ens)$ est le nombre d'occurences de elt dans ens.*

3. Appartenance d'un élément à un multi-ensemble :

   Profil $appartient : \alpha \to multiensemble_\alpha \to \mathbb{B}$

   Sémantique : *$(appartient\ elt\ ens)$ est vrai si et seulement si $elt \in mens$.*

4. Inclusion de deux multi-ensembles :

   Profil $inclus : multiensemble_\alpha \to multiensemble_\alpha \to \mathbb{B}$

   Sémantique : *$(inclus\ ens_1\ ens_2)$ est vrai si et seulement si $ens_1 \subseteq ens_2$*

5. Ajout d'un multi-élément à un multi-ensemble :

   Profil $ajoute : multielement_\alpha \to multiensemble_\alpha \to multiensemble_\alpha$

   Sémantique : *$(ajoute\ elt\ ens)$ est le multi-ensemble obtenu en ajoutant le multi-élément elt au multi-ensemble ens en respectant la contrainte de non répétition de multi-éléments d'une même valeur (revoir la définition de $multiensemble_\alpha$ au début de la section 3).*

6. Suppression d'un multi-élément d'un multi-ensemble :

   Profil $supprime : multielement_\alpha \to multiensemble_\alpha \to multiensemble_\alpha$

   Sémantique : *$(supprime\ (x, n)\ ens)$ supprime n occurrences de l'élément x du multi-ensemble ens. Par convention, si $n = 0$, la fonction est sans effet. Si n est strictement supérieur au nombre d'occurrences de x dans ens, alors toutes les occurrences de x sont supprimées.*

7. Égalité de deux multi-ensembles :

   Profil *egaux* : $multiensemble_\alpha \to multiensemble_\alpha \to \mathbb{B}$

   Sémantique : (*egaux* ens_1 ens_2) *est vrai si et seulement si* ens_1 *et* ens_2 *ont les mêmes multi-éléments.*

8. Intersection de deux multi-ensembles :

   Profil *intersection* : $multiensemble_\alpha \to multiensemble_\alpha \to multiensemble_\alpha$

   Sémantique : (*intersection* ens_1 ens_2) *est le multi-ensemble* $ens_1 \cap ens_2$*, c'est-à-dire le multi-ensemble des éléments appartenant à la fois à* ens_1 *et à* ens_2*.*

9. Différence de deux multi-ensembles :

   Profil *difference* : $multiensemble_\alpha \to multiensemble_\alpha \to multiensemble_\alpha$

   Sémantique : (*difference* ens_1 ens_2) *est le multi-ensemble* $ens_1 \backslash ens_2$*, c'est-à-dire le multi-ensemble des multi-éléments qui appartiennent à* ens_1 *mais pas à* ens_2*.*

10. Obtention aléatoire d'un élément d'un multi-ensemble :

    Profil *un_dans* : $multiensemble_\alpha \to \alpha$

    Sémantique : (*un_dans ens*) *est un élément choisi aléatoirement dans le multi-ensemble ens 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 une seule fois.*

    Indication Il est conseillé d'utiliser la fonction `Random.int` : $\mathbb{N}^* \to \mathbb{N}$; `Random.int(k)` produit aléatoirement un entier compris entre `0` et `k-1`.

    On pourra s'appuyer sur une fonction intermédiaire *ieme* : $\mathbb{N}^* \to multiensemble_\alpha \to \alpha$; (*ieme i ens*) est le i^e élément de *ens*.

3.2 Implémentation des multi-ensembles sur un type somme

Dans cette partie, les $multiensemble_\alpha$ seront implémentés par le type somme récursif polymorphe suivant :

```
type 'a multiensemble =                    (* 'a ≡ α : réservoir des valeurs *)
  | V                                        (* V mis pour « vide » *)
  | A of 'a multielement * 'a multiensemble (* A mis pour « ajout » *)
```

Voici par exemple deux des $3! = 6$ implémentations possibles du multi-ensemble *ex_ens* :

```
let ex_ens1 : nat multiensemble = A((3,2), A((4,1), A((6,3), V)))
and ex_ens2 : nat multiensemble = A((6,3), A((3,2), A((4,1), V)))
```

Q1. Implémenter les fonctions spécifiées dans la section 3.1.

4 Réusinage

24,99 pts

Q2. Réusiner le code de la question précédente en ré-implémentant le type $multiensemble_\alpha$ grâce aux listes natives d'OCaml :

```
type 'a multiensemble = 'a multielement list
```

Q3. Après avoir abordé en cours, TD ou TP le chapitre sur l'ordre supérieur, réusiner à nouveau les fonctions suivantes – spécifiées section 3.1 – grâce aux fonctions du module OCaml `List` *fold* (`_left` ou `_right`), `map`, `for_all`, `exists`, `filter`, ... :
a) *cardinal*, b) *nbocc*, c) *appartient*, d) *inclus*, e) *ajoute*, f) *supprime*, g) *intersection*, h) *difference*.

Remarque Il est posible de poursuivre cet énoncé sans avoir répondu à cette question.

5 Le 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). Par exemple, la figure 1 ci-contre montre trois tuiles de valeur 12 : une bleu, une noire et une rouge.

FIG. 1: trois tuiles

Le jeu comporte deux séries de telles tuiles. Ainsi, chaque tuile avec une certaine valeur apparaît deux fois pour chaque couleur et il y a deux jokers. Cela fait donc en tout 106 tuiles :

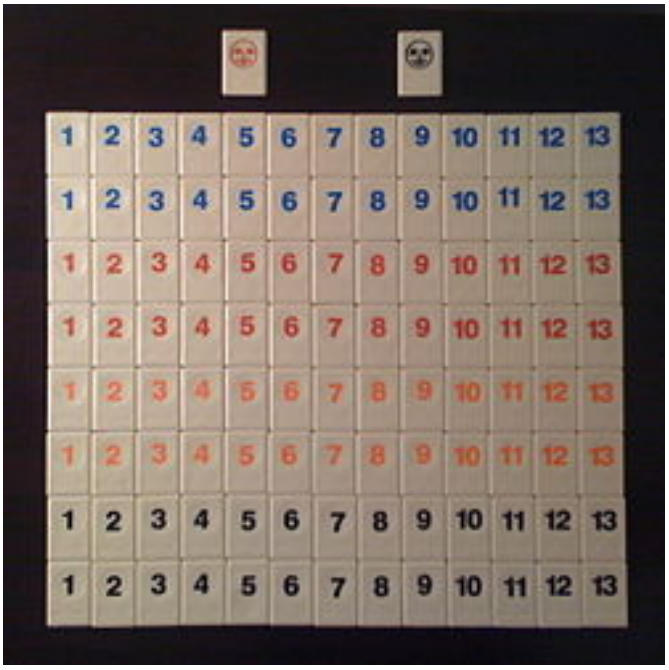

FIG. 2: les 106 tuiles du jeu

5.1 Règles du jeu

Les règles décrites dans ce projet sont une simplification de celles du jeu original (voir par exemple `https://fr.wikipedia.org/wiki/Rummikub` pour une description complète). De plus, on ne s'intéresse ici qu'au jeu à deux joueurs.

5.1.1 Règle 1 : constitution de la table de jeu

Chaque joueur dispose d'un certain nombre de tuiles qu'il est le seul à voir, et qui forment sa *main*. Sur la figure 3 ci-dessous, les mains des joueurs sont disposées sur des chevalets bleu.

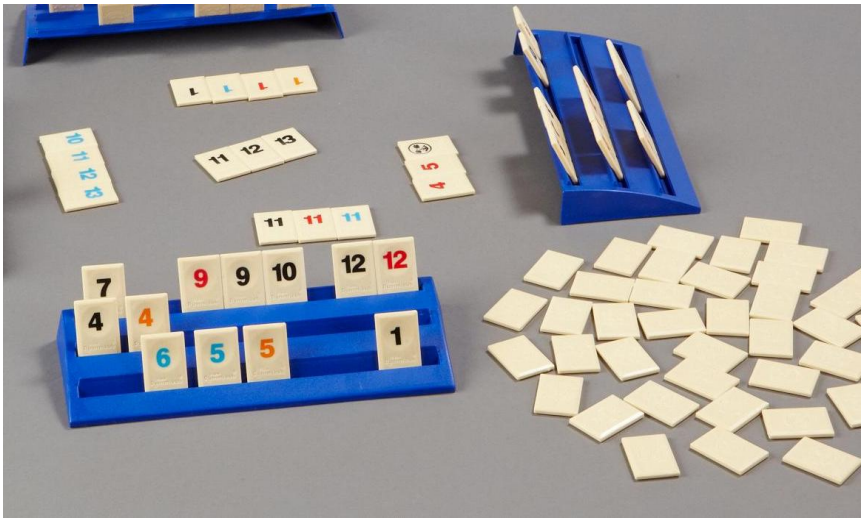

FIG. 3: une table de jeu

Au centre de la *table* sont disposées les tuiles déjà en jeu (en haut à gauche sur la figure). Elles sont regroupées par paquets. Chaque paquet doit former une combinaison valide. Les combinaisons valides sont décrites dans la règle 4. Sur la table de jeu de la figure 3 ci-dessus, 17 tuiles formant 5 combinaisons ont été posées par les joueurs.

Les tuiles qui ne sont ni en jeu dans des combinaisons, ni dans les mains des joueurs, constituent la *pioche*. Les tuiles de la pioche ne sont pas visibles (en bas à droite de la figure 3).

5.1.2 Règle 2 : distribution initiale

Au début du jeu, il n'y a pas de tuile sur la table. La distribution initiale, 14 tuiles par joueur, est composée de tuiles tirées au hasard parmi les tuiles de la pioche.

Le score initial de chaque joueur est 0.

5.1.3 Règle 3 : tour de jeu

Quand vient son tour, un joueur peut :

- soit piocher, c'est-à-dire tirer au hasard une tuile de la pioche et la mettre dans sa main ;
- soit poser un nombre non nul des tuiles de sa main sur la table en réorganisant les tuiles en jeu pour que les nouvelles combinaisons soient valides (sauf pour la première pose, voir règle 5).

À l'issue d'une de ces deux actions, le tour du joueur est terminé.

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.

5.1.4 Règle 4 : combinaisons valides

Les combinaisons valides sont :

- Les *suites* monochromes d'au moins 3 tuiles consécutives.

  Exemple : cinq tuiles rouges de valeur 2, 3, 4, 5 et 6.

- Les *groupes* de trois ou quatre tuiles de couleurs distinctes et comportant le même entier.

  Exemple : trois tuiles de valeur 12 de couleur rouge, jaune et bleu.

5.1.5 Règle 5 : première pose

Lors de sa première pose, un joueur doit poser une ou plusieurs combinaisons valides dont la valeur totale (somme des valeurs des tuiles) est au moins 30. Le joker compte pour l'entier qu'il remplace.
Il est interdit de modifier les combinaisons déjà en place lors de cette première pose.

5.1.6 Règle 6 : arrêt du jeu

Le jeu s'arrête dès qu'un joueur a vidé sa main, ou si aucun des joueurs ne peut plus jouer : ils ne peuvent plus piocher car la pioche est vide et ils ne peuvent plus poser de tuiles.

5.1.7 Règle 7 : calcul du score et joueur gagnant

On ne marque pas de point au cours de la partie. Au moment où le jeu se termine, le score d'un joueur est égal à la somme des valeurs des tuiles présentes dans sa main.
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.

6 Modélisation des données

10 pts

6.1 Les tuiles

Une tuile est soit un joker, soit caractérisée par une valeur et une couleur. On définit donc :

$$tuile \stackrel{def}{=} \{Joker\} \cup \{T(v,c) \ / \ v \in valeur,\ c \in couleur\}$$

où *valeur* est l'intervalle de $\mathbb{N}$ $\{1, \dots, 13\}$ et *couleur* est l'énumération des couleurs du jeu (bleu, rouge, jaune et noir).

Q4. Implémenter *couleur*, *valeur* et *tuile*.

6.2 Combinaisons, tables et poses

Une *combinaison* est une séquence de tuiles. La *table* est constituée par la séquence des combinaisons déjà posées. Une *pose* est également une séquence de combinaisons.

Q5. Implémenter *combinaison*, *table* et *pose*.

Dans une combinaison, l'ordre des tuiles peut être significatif, par exemple dans le cas d'une suite (règle 4). L'ordre des combinaisons d'une table ou d'une pose n'est pas significatif.

6.3 Mains et pioches

Les mains des joueurs ainsi que la pioche sont des muti-ensembles de tuiles.

Q6. Implémenter *main* et *pioche*.

Les 106 tuiles du jeu – la pioche initiale – seront représentées par le multi-ensemble défini ci-dessous :

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

Pour l'affichage des mains des joueurs et de la pioche (voire de la table), il est commode de présenter les tuiles toujours dans le même ordre. On définit un ordre (dit lexicographique) sur les tuiles de la façon suivante :

- les jokers sont strictement supérieur à toutes les autres tuiles ;
- bleu < rouge < jaune < noir ;
- au sein d'une même couleur, $1 < 2 < ... < 13$.

D'où la spécification de la fonction :

Profil *en_ordre* : $multiensemble_{tuile} \rightarrow multiensemble_{tuile}$
Sémantique : *en_ordre(ens) est la représentation ordonnée du multi-ensemble de tuiles ens selon l'ordre défini ci-dessus.*

Q7. Implémenter *en_ordre*.

6.4 Les joueurs

Le *statut* d'un joueur est constitué des informations suivantes :

- son identifiant : `J1` pour un des deux joueurs, `J2` pour l'autre ;
- son engagement dans la partie : vrai s'il a déjà posé des tuiles (c'est-à-dire si ce n'est pas son premier coup), faux sinon ;
- sa main.

On implémentera ces notions grâce aux types :

```
type joueur = J1 | J2
type statut = joueur * bool * main
```

6.5 État d'une partie

L'*état* d'une partie est constitué des informations suivantes :

- les statuts des joueurs ;
- les combinaisons en jeu ;
- la pioche ;
- l'identifiant du joueur dont c'est le tour.

On implémentera ces notions grâce au type :

```
type etat = (statut * statut) * table * pioche * joueur
```

Soit $((s_1, s_2), \_, \_, \_)$ un état ; s_1 est le statut de `J1`, s_2 est le statut de `J2`.

Par exemple, lors du premier tour, `J1` a posé les deux combinaisons (7, rouge), (7, Jaune), (7, jaune), (1, noir) et (2, noir), (3, noir), (4, noir) ; `J2` a posé la combinaison (9, rouge), (10, rouge), (11, rouge). L'état du jeu est alors :

- statuts des joueurs :
  - `J1`, vrai, tuiles restantes dans la main de `J1`
  - `J2`, vrai, tuiles restantes dans la main de `J2`
- combinaisons en jeu :

  (7, rouge) (7, Jaune) (7, jaune)

  (1, noir) (2, noir) (3, noir) (4, noir)

  (9, rouge) (10, rouge) (11, rouge)
- pioche : tuiles de la pioche
- à qui le tour : `J1`.

6.5.1 État initial

Au début de la partie, chaque joueur reçoit 14 tuiles tirées au hasard dans la pioche initiale. Pour réaliser cette opération, on spécifie :

1. Extraction de tuiles d'une pioche :

   Profil *extraire* : $\mathbb{N} \to pioche \to main \times pioche$

   Sémantique : *(extraire n p) est un couple (m, nlle_p) où m est une main de n tuiles tirées au hasard dans p, et nlle_p est la nouvelle pioche.*

2. Distribution :

   Profil *distrib* : $\{\} \to main \times main \times pioche$

   Sémantique : *À partir de la pioche initiale, distrib() génère une main pour chacun des deux joueurs et la pioche restante.*

3. État initial :

   Profil *init_partie* : $\{\} \to etat$

   Sémantique : *init_partie() définit un état initial de la partie, après distribution.*

Les fonctions *distrib* et *init_partie* sont des fonctions sans paramètre : elles n'ont pas besoin de données. Leur profil indique donc un ensemble de départ vide : $\{\}$. En OCaml, $\{\}$ est implémenté par le type `unit`. Ce type ne contient qu'une seule valeur, notée `()` ; cette syntaxe est cohérente avec les appels de fonctions sans argument : *distrib*(), *init_partie*().

Q8. Implémenter les fonctions ci-dessus.

Remarque Il est possible de poursuivre cet énoncé sans avoir répondu à cette question. Dans ce cas, il faudra définir un état initial « à la main ».

6.5.2 Accès aux informations d'un état

Il peut être utile de définir des fonctions d'accès aux différents informations de l'état, comme par exemple :

1. *joueur_courant*, *joueur_suivant* : *etat* → *joueur*
2. *la_table* : *etat* → *table*
3. *la_pioche* : *etat* → *pioche*
4. *le_statut* : *joueur* → *etat* → *statut*
5. *la_main* : *joueur* → *etat* → *main*

Q9. Implémenter les fonctions d'accès ci-dessus.

7 Mise en œuvre des règles

15 pts

Les parties 7.1 et 7.2 peuvent être traitées dans n'importe quel ordre.

7.1 Validité des combinaisons

Afin de pouvoir tester la validité des combinaisons (règle 4), nous spécifions les prédicats suivants :

1. Suite :

   Profil *est_suite* : *combinaison* $\rightarrow \mathbb{B}$

   Sémantique : *est_suite*(c) *est vrai si et seulement si la combinaison c est une suite.*

2. Groupe :

   Profil *est_groupe* : *combinaison* $\rightarrow \mathbb{B}$

   Sémantique : *est_groupe*(c) *est vrai si et seulement si la combinaison c est un groupe.*

3. Validité d'une combinaison :

   Profil *combinaison_valide* : *combinaison* $\rightarrow \mathbb{B}$

   Sémantique : *combinaison_valide*(c) *est vrai si et seulement si la combinaison c est valide.*

4. Validité d'une séquence de combinaisons :

   Profil *combinaisons_valides* : *combinaison list* $\rightarrow \mathbb{B}$

   Sémantique : *combinaisons_valides*(l) *est vrai si et seulement si les combinaisons de la liste l sont valides.*

Q10. Implémenter les fonctions spécifiées ci-dessus.

Remarque L'utilisation de l'ordre supérieur, lorsque c'est possible, sera appréciée.

7.2 Calcul de points

Afin de pouvoir calculer les scores des joueurs, nous spécifions les fonctions suivantes :

1. Points d'une suite :

   Profil *points_suite* : *combinaison* $\rightarrow \mathbb{N}$

   Sémantique : *points_suite*(s) *est le nombre de points de la suite s.*

2. Points d'un groupe :

   Profil *points_groupe* : *combinaison* $\rightarrow \mathbb{N}$

   Sémantique : *points_groupe*(g) *est le nombre de points du groupe g.*

3. Points d'une pose :

   Profil *points_pose* : *pose* $\rightarrow \mathbb{N}$

   Sémantique : *points_pose*(p) *est le nombre de points total des combinaisons de la pose p.*

Q11. Implémenter les fonctions spécifiées ci-dessus.

8 Gameplay

25,01 pts

Pour pouvoir utiliser les fonctions spécifiées dans la partie 3, il est nécessaire de convertir les tables en multi-ensembles de tuiles.

Q12. Implémenter la fonction suivante :

Profil *tableVmens* : $table \to multiensemble_{tuile}$
Sémantique : *tableVmens*(t) *est le multiensemble des tuiles de la table* t.

Remarque Ici encore, l'utilisation de l'ordre supérieur sera appréciée.

8.1 Validité des coups

On distingue les coups du premier tour de jeu des autres coups :

1. Validité du premier coup :

   Profil *premier_coup_ok* : $main \to pose \to main \to \mathbb{B}$
   Sémantique : (*premier_coup_ok* $m_0\ p_0\ m_1$) *est vrai si et seulement si à partir de la main* m_0, *la proposition de combinaisons à poser* p_0 *et la nouvelle main* m_1 *constituent un coup valide pour un premier tour.*

2. Validité des autres coups :

   Profil *coup_ok* : $table \to main \to table \to main \to \mathbb{B}$
   Sémantique : (*coup_ok* $t_0\ m_0\ t_1\ m_1$) *est vrai si et seulement si à partir de la table* t_0 *et de la main* m_0, *il est possible de produire la table* t_1 *et la main* m_1.

Q13. Implémenter les fonctions ci-dessus.

8.2 Ajout d'une tuile sur la table

Q14. Implémenter la fonction suivante :

Profil *ajouter_tuile* : $table \to tuile \to table$
Sémantique : (*ajouter_tuile t tbl*) *est la table obtenue en ajoutant la tuile* t *à l'une des combinaisons de la table tbl* (*dont toutes les combinaisons sont valides*) *si cela est possible, ou la table vide sinon.*

8.3 Recherche de combinaisons dans une main

Les combinaisons valides étant les suites et les groupes, on spécifie les deux fonctions suivantes :

1. Extraction d'une combinaison :

   Profil *extraction_suite, extraction_groupe* : $main \to combinaison$
   Sémantique : *extraction_suite*(m) (*respectivement extraction_groupe*(m)) *extrait une suite* (*resp. un groupe*) *de la main* m.

Q15. Implémenter les fonctions ci-dessus selon l'un des deux algorithmes suivants :

Algorithme 1. Extraire au hasard un certain nombre de tuiles de la main et vérifier si elles forment une suite (resp. un groupe). Si ce n'est pas le cas, essayer à nouveau. Le nombre d'essais est à déterminer.

Algorithme 2. Énumérer toutes les combinaisons possibles que l'on peut former avec les tuiles de la main m. Peut-on évaluer ce nombre de combinaisons ?

8.4 Changement d'état

Les changements d'état se produisent lorsque un joueur pioche ou joue un coup :

1. Changement d'état lors d'une pioche :

   Profil *piocher* : *etat* → *etat*

   Sémantique : *piocher*(e) *est l'état obtenu à partir de l'état e après que le joueur courant ait pioché une carte. Le tour passe au joueur suivant.*
   Si la pioche est vide, *piocher*$(e) = e$.

2. Changement d'état lors d'un coup :

   Profil *jouer_1_coup* : *etat* → *table* → *etat*

   Sémantique : (*jouer_1_coup e tbl*) *est l'état obtenu à partir de l'état e avec la proposition de table tbl.*
   S'il s'agit du premier coup, (*jouer_1_coup e _*) $= e$.

   Exemple :

   Soit *e* l'état courant suivant :

   - statuts des joueurs :
     - `J1`, vrai, ((7, noir), 2) ((8, rouge) 1) ((3, noir), 1) ((5, noir), 2)
     - `J2`, vrai, tuiles restantes dans la main de `J2`
   - combinaisons en jeu :
     (7, rouge) (7, bleu) (7, jaune)
     (1, noir) (2, noir) (3, noir) (4, noir)
     (9, rouge) (10, rouge) (11, rouge)
   - pioche : tuiles de la pioche
   - à qui le tour : `J1`

   Soit *tbl* la table proposée 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 tbl*) est correct et produit l'état suivant :

- statuts des joueurs :
  - `J1`, vrai, $((7, \text{noir}), 1)$ $((5, \text{noir}), 1)$
  - `J2`, vrai, tuiles restantes dans la main de `J2`
- combinaisons en jeu :
  $(7, \text{rouge})$ $(7, \text{bleu})$ $(7, \text{jaune})$ $(7, \text{noir})$
  $(1, \text{noir})$ $(2, \text{noir})$ $(3, \text{noir})$
  $(3, \text{noir})$ $(4, \text{noir})$ $(5, \text{noir})$
  $(8, \text{rouge})$ $(9, \text{rouge})$ $(10, \text{rouge})$ $(11, \text{rouge})$
- pioche : tuiles de la pioche
- à qui le tour : `J2`

3. Changement d'état lors du premier coup :

   Profil *jouer_1er_coup* : *etat* → *pose* → *etat*

   Sémantique : (*jouer_1er_coup e p*) *est l'état obtenu à partir de l'état e avec la proposition de pose p.*
   S'il ne s'agit pas du premier coup ou si p est incorrecte (par exemple si la somme des points est inférieure ou égale à 30) (*jouer_1er_coup e _*) = *e*.

Q16. Implémenter les changements d'états.

8.5 Scenario de jeu

Voici un exemple de scenario de jeu :

0. Début de partie, on (ré)initialise l'état initial :

```
let etat_init : etat = init_partie()
```

1. `J1` joue son premier coup :

```
let e1 : etat = jouer_1er_coup etat_init
                                [ [T(2,Jaune) ; T(3,Jaune) ; T(4,Jaune)] ;
                                  [T(7,Jaune) ; T(8,Jaune) ; Joker] ]
```

2. Puis `J2` et `J1` piochent alternativement :

```
let e2 : etat = piocher e1
let e3 : etat = piocher e2
let e4 : etat = piocher e3
let e5 : etat = piocher e4
let e6 : etat = piocher e5
```

3. `J1` pose des tuiles sur la table :

```
let e7 = jouer_1_coup e6 [ [T(2,Jaune) ; T(3,Jaune) ; T(4,Jaune)] ;
                           [T(7,Jaune) ; T(8,Jaune) ; Joker ; T(10,Jaune)] ]
```

4. `J2` joue son premier coup :

```
let e8 = jouer_1er_coup e7 [ [Joker ; T(11,Rouge) ; T(11,Bleu)] ]
```

5. `J1` pioche :

```
let e9 = piocher e8
```

6. J2 pose des tuiles sur la table :

```
let e10 = jouer_1_coup e9
            [ [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)] ]
```

7. ...

Q17. Proposer un autre scenario de jeu dont les coups et les enchaînements seront différents de ceux de l'exemple ci-dessus.

9 Pour aller plus loin

hors barème

Jusqu'à maintenant, les fonctions qui ont été définies constituent ce qu'on pourrait qualifier de *noyau fonctionnel* du jeu : elles ne manipulent que des représentations internes des données, peu adaptées à un utilisateur humain.

Ainsi, même s'ils permettent d'effectuer des tests, les scénarii comme le précédent sont loins d'être optimaux en terme de jouabilité, notamment à cause de l'abscence d'une réelle *interface persone-machine* (IPM).

Un moteur de jeu réaliste devrait a minima permettre d'enchaîner les coups jusqu'à la fin de la partie en faisant jouer alternativement les joueurs (ou bien un seul joueur contre l'ordinateur) via une interface aussi ergonomique que possible, quand bien même en mode texte : saisie des tables successives au clavier, affichage textuel des états de manière claire et lisible.

Q18. *** Développer un moteur de jeu accompagné d'une IPM textuelle.

Q19. ***** Réusiner le moteur de la question précédente pour que l'IPM soit graphique.

On pourra utiliser une des librairies suivantes :

1. *Bogue*, basée sur la SDL2 (Simple Directmedia Layer).
   `https://github.com/sanette/bogue`
2. *Brisk*, développée en Reason qui réunit les écosystèmes OCaml et Javascript
   `https://github.com/briskml/brisk`
3. *Revery*, également développée en Reason
   `https://github.com/revery-ui/revery`