Algorithmique et Programmation Fonctionnelle

Projet : Variations sur le rami

Objectifs

Le but de ce projet est de programmer les différentes variantes du rami qui utilisent des lettres (rami des lettres), des chiffres (*Rummikub*) ou des cartes (rami ou canasta). Ces variantes peuvent être vues comme l'application d'un même foncteur (de jeu) à plusieurs modules (de règles).

Une partie importante du projet concerne la manipulation de structures de données : nous devrons gérer des ensembles (de tuiles ou de cartes), et vérifier qu'un mot joué est présent dans un dictionnaire. D'autre part, nous aurons différentes régles de comptage des scores en fonction des différentes versions, ce qu'on réalise à l'aide du système de modules. Enfin il faudra sauvegarder et charger une partie entre plusieurs joueurs, ce qui met en œuvre l'analyse syntaxique.Les plus courageux pourront réaliser une intelligence artificielle afin de jouer contre la machine.

Rappelons enfin le principe général d'un projet. Il ne s'agit pas comme dans certains examens de traiter des questions sans rapport un peu partout pour récupérer un maximum de points, mais au contraire d'avoir un produit fini qui fonctionne, quitte à ce que certaines fonctionnalités en soient absentes. À vous donc de déterminer quelles parties de ce sujet vous semblent prioritaires, et en cas de difficultés sur un point, s'il vaut mieux insister pour le résoudre ou l'abandonner complètement.

Quelques fichiers de départ vous sont fournis, ainsi que des compléments en cas de besoin [1].

Travail demandé

Le barème est donné à titre indicatif, notamment pour vous permettre d'anticiper le volume de travail que représente chaque module. Le TP en entier est à rendre par courriel à tous les enseignants, impérativement avant le *19 décembre 2014 minuit* sous la forme d'un fichier `noms.tar.gz`. Seuls les TP respectant les contraintes suivantes seront corrigés :

1. `noms` correspond aux noms des membres du groupe
2. la commande `tar xzf nom.tar.gz` génère un répertoire `noms` qui contiendra **obligatoirement** : les sources de votre application, un fichier `Makefile`, un fichier `README` (et rien d'autre).
3. la commande `make` génère un fichier exécutable (ou plusieurs suivant les jeux que vous avez implémentés) `main.native` ou `main`.

Par ailleurs, le fichier `README` doit contenir :
— le principe du TP en une dizaine de lignes ;
— exercice par exercice, la liste de ce qui marche et ce qui ne marche pas ;
— si vous vous êtes fait aider, par qui et à quel endroit ;
— comment utiliser votre programme, quels fichiers il faut charger, les différents exemples de votre projet que vous avez programmés, etc.

Dans les fichiers d'interface, et directement en entête de chaque fonction, vous préciserez :
— le type de la fonction ;
— le rôle de la fonction et à quoi correspondent ses entrées et ses sorties ;
— si besoin est, un commentaire sur les choix d'implémentation que vous avez faits, et plus généralement toute information susceptible d'aider à la bonne lecture du code.

Rappel : la note de contrôle continu est constituée de la note de projet (50%), de la note de DM (30%) et des TPs (20%, automatiquement acquis si tous les TPs sont rendus).

1. `http://www-verimag.imag.fr/~wack/APF_2014_2015/fichiers_projet.tar.gz`

1 Vue d'ensemble : les règles communes

Nous présentons d'abord les parties communes dans les règles des différentes variantes de rami. Cela nous permet de définir une signature `REGLE` que nous spécialiserons en différents modules (`Lettres`, `Rummikub`, ...).

Les règles communes sont les suivantes :

1. Le jeu se joue avec des tuiles comportant une valeur (type `t` dans la signature). Par exemple dans le Rummikub une valeur est un couple (nombre, couleur). Les tuiles du jeu sont en nombre fini et sont représentées par un multi-ensemble nommé `paquet`.
2. 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 groupes 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*. Le type des combinaisons est nommé `combi` et il est imposé égal à `t list`. Le type d'une main, et de la pioche, est `main`.
3. Au début du jeu, il n'y a pas de tuile en jeu et la main de chaque joueur est formée d'un certain nombre de tuiles (le même pour chaque joueur, nommé `main_initiale` dans la signature) tirées au hasard parmi les tuiles de la pioche. Le score initial de chaque joueur est 0.
4. À 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, et de respecter la règle particulière des jokers (cf bonus 1). La fonction `combi_valide` détermine si une combinaison est valide. Dans le cas où la pioche est vide et le joueur souhaite piocher, le tour passe au joueur suivant sans modification du jeu ni de la main du joueur.
5. La première fois qu'un joueur souhaite poser des tuiles de sa main, il ne peut pas modifier les combinaisons déjà en place : il ne peut qu'ajouter de nouvelles combinaisons à partir de tuiles provenant de sa main uniquement. Il existe de plus des contraintes supplémentaires propres à chaque variante et la fonction `premier_coup_valide` permet de tester si un coup est valide pour une première pose.
6. Après son tour, un joueur peut être contraint de piocher des tuiles pour ramener sa main à une taille déterminée par la variante particulière de rami que l'on considère et nommé `main_min`.
7. Un joueur qui pose peut marquer des points (cela dépend de la règle utilisée). Ces points dépendent des tuiles posées en jeu et des combinaisons formées par le joueur. Le calcul des points gagnés lors d'un coup est effectué par la fonction `points`.
8. La partie se termine dès qu'un joueur vide sa main lors de son tour, avec pour certaines variantes la condition supplémentaire que la pioche restante soit vide (dans ce cas le booléen `fin_pioche_vide` vaut `true`).
9. À la fin de la partie, chaque joueur peut éventuellement marquer des points en fonction de sa main restante. On les calcule avec la fonction `points_finaux`.
10. Un certain nombre des tuiles disponibles sont des jokers et peuvent prendre la valeur de n'importe quelle autre tuile lors de la formation des combinaisons.

La signature `REGLE` vous est fournie. Comme vous pouvez le voir, nous aurons besoin de manipuler des multi-ensembles (pour toutes les variantes). De plus pour la variante se jouant avec des lettres, nous aurons besoin de savoir gérer un dictionnaire. Ces modules complémentaires sont l'objet des deux sections suivantes.

La fonction `lit_valeur` permet de réaliser l'analyse syntaxique d'une valeur (parser). Réciproquement, `ecrit_valeur` permet de convertir une valeur en chaîne de caractères. Cela sera utile pour les fonctionnalités de sauvegarde/chargement de partie.

2 Module multi-ensembles (2 points)

Un multi-ensemble contient des éléments (d'un même type) et peut contenir plusieurs exemplaires d'un même élément (on parle de la *multiplicité* d'un élément). La différence avec une liste est qu'un multi-ensemble ne tient pas compte de l'ordre de ses éléments : $\langle 1, 2, 1\rangle = \langle 2, 1, 1\rangle$.

Vous êtes libres de représenter cette structure de données de la façon qui vous semble adéquate. Le module que vous écrirez sera forcément polymorphe (on pourra manipuler des multi-ensembles d'entiers, de nombres flottants, de chaînes de caractères...). Quelques éléments suggérés de la signature :

— une valeur `nil` pour représenter le multi-ensemble vide ;
— une fonction d'union de deux multi-ensembles ;
— un test d'appartenance à un multi-ensemble ;
— un test d'égalité entre deux multi-ensembles.

Exercice 1 *Écrire un module `MultiEnsemble` qui définira le type d'un multi-ensemble polymorphe. Le type d'un multi-ensemble d'objets de type `'a` sera nommé `'a mset`. Votre module sera amené à être complété au fur et à mesure de votre progression dans le jeu de rami.*

3 Module Dictionnaire (3 points)

Il existe plusieurs solutions afin de représenter un dictionnaire. Nous choisissons celle par un arbre 26-aire, ce qui signifie que l'on considère les mots constitués des 26 lettres de l'alphabet français sans tenir compte des accents, cédiles... Chacun des 26 liens d'un nœud à l'un de ses fils correspond à une lettre de l'alphabet, et chaque nœud de l'arbre sera étiqueté par un booléen indiquant si le mot obtenu en parcourant l'arbre de la racine jusqu'à ce nœud appartient au dictionnaire ou pas.

Vous regrouperez les définitions de type et de fonctions liés au dictionnaire dans un module nommé `Dictionnaire`. Le type du dictionnaire est imposé (voir fichier `.mli` fourni).

Indications : pour créer un dictionnaire vide on écrira

```
let dico_vide = Noeud((Array.make 26 Feuille), false)
```

Il est fortement conseillé de consulter la documentation OCaml à propos des tableaux (`Array`) et des chaînes de caractères (`String`) avant de se lancer dans cette partie.

Exercice 2 *Représentation et fonctionnalités*

— *Implémenter la fonction `member`, qui retourne un booléen indiquant la présence ou non d'un mot dans le dictionnaire. Attention : votre mot en entrée sera forcément de type `string` et vous devrez tenir compte du joker que l'on représentera par le caractère `*`*[2]*. Par exemple si votre dictionnaire $\mathbb{D}$ contient le mot `cal`, vous devrez accepter le mot `c*l` comme appartenant à $\mathbb{D}$.*
— *Implémenter la fonction `add`, qui ajoute un mot dans le dictionnaire.*
— *Implémenter la fonction `remove`, qui retire un mot du dictionnaire s'il est présent et ne fait rien sinon.*
— *Implémenter une fonction `of_stream` qui crée un dictionnaire à partir d'un flux contenant un mot par ligne.*
— *Implémenter une fonction `to_list` qui convertit un dictionnaire en sa liste de mots.*
— *Fournir un jeu de test pour votre module dictionnaire.*
— *Tester cela sur un vrai dictionnaire comme par exemple celui proposé : `http://www.pallier.org/ressources/dicofr/dicofr.html`*

Bonus (2 points) Proposer une implémentation d'un dictionnaire reposant sur une autre structure de données. Ce bonus ne dispense pas de réaliser la première version des dictionnaires. [3] [4]

2. Attention : le joker n'apparait en aucun cas dans le dictionnaire mais uniquement dans les mots à tester par `member`.

3. [1] Andrew W. Appel and Guy J. Jacobson. The World's Fastest Scrabble Program. Communications of the ACM, 31(5) :572–578, 1988.

4. [2] Steven A. Gordon. A Faster Scrabble Move Generation Algorithm. Software, Practice and Experience, 24(2) :219–232, 1994.

4 Sauvegarde et lecture depuis un fichier (4 points)

Il est demandé de pouvoir sauvegarder l'état d'une partie, et de pouvoir la relire ensuite pour la reprendre. L'état d'une partie est constituée des informations suivantes :

— les noms des joueurs, leur score, s'ils ont déjà posé ou non, leur main ;
— les combinaisons en jeu ;
— la pioche restante ;
— le numéro du joueur dont c'est le tour.

On demande une représentation de la forme suivante :

```
(joueurs
 (Pascal 17 true (S P O I N E L *))
 (Laurent 42 true (N S A V U L G I O))
 (Marion 0 false (E E I N M Z S O O E V C N L)))
(jeu
 (F A C I L E)
 (E X A M E N)
 (C A * T O N)
 (E L E V E)
 (B R I L L A N T))
(pioche
 C S N O H I N U A E L T M X)
(tour 3)
```

Dans cette partie Pascal et Laurent ont déjà posé, ce qui n'est pas le cas de Marion, et le prochain joueur à jouer est Marion. Le nombre de blancs (espaces, retours à la ligne) entre chaque token est arbitraire, l'exemple est ici aéré pour plus de lisibilité. Dans ce format, seul la représentation des valeurs des tuiles dépend de la variante utilisée, le reste est commun.

Pour simplifier la lecture d'une partie sauvegardée, on définit le type `token` suivant :

```
type token = LPar | RPar | TGen of string
```

et on impose que la représentation d'une valeur du jeu, peu importe la règle considérée, soit :

— soit un atome, c'est-à-dire une chaîne (token `TGen`) ne contenant pas de parenthèse ;
— soit une liste bien parenthésée de `token`.

Dans chacun des modules implémentant la signature `REGLE` que vous écrirez, il y aura une fonction `lit_valeur` de type `token list -> t`.

Remarque : pour utiliser les constructions `parser` sur les flots, il faut rajouter l'option `-pp "camlp4o.opt -unsafe"` à `ocamlbuild` pour la compilation.

L'implémentation des fonctions relative à la sauvegarde et au chargement des parties sera demandée en section 7.

5 Le rami des lettres (4 points)

Règles : Nous présentons d'abord le rami des lettres. Par rapport aux règles générales données au début, le rami des lettres a les particularités suivantes :

1. Les combinaisons autorisées sont les mots du dictionnaire fourni à l'url précédente [5] de longueur 3 lettres au moins et ne contenant aucun autre caractère que les 26 lettres de l'alphabet français sans accent.
2. Lors de la réorganisation des tuiles en jeu, il est interdit de faire apparaître plusieurs fois un même mot.

5. Si vous n'avez pas fini l'implantation du dictionnaire vous pouvez ignorer cette règle dans un premier temps et accepter toute combinaison de 3 lettres au moins.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
8	2	3	3	16	2	2	2	9	1	1	6	4	7	7	2	1	7	7	7	7	2	1	1	1	1

FIGURE 1 – La répartition des lettres au rami des lettres.

3. La pioche initiale est donnée dans le tableau 1 à laquelle s'ajoutent 2 jokers.
4. Au début du jeu, on distribue à chaque joueur 14 lettres.
5. Lors de sa première pose, un joueur doit poser un mot de 6 lettres au moins. Les lettres de ce mot doivent toutes provenir uniquement du stock de lettres du joueur. La pose de ce premier mot termine le tour du joueur.
6. Le joueur doit toujours avoir au moins 7 lettres dans son stock, il tire donc de nouvelles lettres une fois son tour fini si nécessaire. Si la pioche ne contient plus assez de lettres pour compléter son stock à 7 lettres, le joueur ramasse autant de lettres que possible et la pioche est épuisée.
7. À son tour, un joueur marque un point par lettre de sa main posée en jeu. Ces points sont doublés si le joueur parvient à vider sa main à ce tour. De plus le joueur marque un point par lettre du mot le plus long qu'il a formé (c'est-à-dire un mot qui n'apparaissait pas avant son tour).
8. Le jeu se termine lorsque la pioche est épuisée et qu'un joueur vide sa main.

Exemple : Un joueur possède les lettres `R`, `O`, `S` et les mots `DROLE` et `BOEUF` sont posés sur la table. Le joueur peut poser ces trois lettres et former les mots suivants : `ROLE`, `BORDS` et `OEUF`. Le joueur marque alors 3 points (3 lettres placées) plus 5 points (mots le plus long contenant des nouvelles lettres) soit un total de 8 points.

Exercice 3 *Écrire le module* `Lettres` *respectant la signature* `REGLE`*.*

6 Rummikub (4 points)

Nous décrivons maintenant les règles propres au Rummikub.

1. Les valeurs possibles des tuiles sont des entiers de 1 à 13 associés à une couleur (bleu, rouge, jaune ou noir), ou un joker.
2. Dans la pioche initiale, chaque entier de 1 à 13 apparaît deux fois pour chaque couleur et il y a deux jokers. Cela fait donc 106 tuiles.
3. La distribution initiale est de 14 tuiles par joueur.
4. Les combinaisons valides sont :
   — Les suites monochromes d'au moins 3 tuiles consécutives, par exemple `4-5-6-7` ;
   — Les groupes d'au moins trois tuiles de couleurs distinctes et comportant le même entier, par exemple trois tuiles `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.
6. Le jeu s'arrête dès qu'un joueur a vidé sa main.
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.

Exercice 4 *Écrire le module* `Rummikub` *respectant la signature* `REGLE`*.*

7 Jouer (3 points)

Nous avons maintenant deux implémentations des règles. Il est temps de les mettre en pratique à l'aide d'un foncteur `Jeu`. Le foncteur que vous écrivez devra se conformer à la signature `TJeu` qui vous est fournie :

1. `coup_valide` détermine si un coup est valide. Les arguments sont le jeu en cours, la main du joueur, le nouveau jeu après ce coup, la nouvelle main du joueur et un booléen indiquant si le joueur a posé ou non.
2. `initialiser` crée un état initial du jeu à partir d'une liste de noms de joueurs, en distribuant un main de départ à chacun, etc.
3. `lit_coup` interagit avec l'utilisateur pour lui demander un coup à jouer, ou s'il pioche. Les arguments sont le nom du joueur dont c'est le tour, le jeu en cours, la main du joueur, et un booléen indiquant si le joueur a déjà posé. Vous gérerez l'interaction au clavier pour lire le coup du joueur, et vous bouclerez tant que l'utilisateur n'a pas rentré un coup valide [6].
4. `sauvegarde` convertit la partie en cours en une chaîne de caractères suivant le format indiqué en section 4.
5. `chargement` lit une partie depuis un flux en suivant le même format.
6. `joue` conduit une partie entre joueurs humains, de l'état passé en argument jusqu'à la fin de la partie. La sortie est la liste des scores des joueurs. Il peut être utile d'indiquer à chaque étape le joueur dont c'est le tour, les scores de tous les joueurs...

Exercice 5 *Réaliser ce foncteur.*

Indication : une utilisation raisonnée des traits impératifs d'OCaml facilitera la réalisation de ce foncteur.

Bonus 1 : Règle des jokers (1 point)

Implémenter la règle complète liée aux jokers, qui s'énonce ainsi : lors de la réorganisation des tuiles en jeu, une combinaison contenant au moins un joker est intouchable (invariante) tant que le joueur n'aura pas remplacé un à un tous les jokers composant cette combinaison par des tuiles provenant de sa main. Les jokers obtenus par ce remplacement doivent être recombinés dans le jeu dans ce même tour.

Bonus 2 : Majhong, Rami des cartes, Canasta (2 points)

Comment nous l'avons préciser au début de ce TP, il existe plusieurs variantes de ce jeu.

Exercice 6 *Implémenter une seul variante de plus de ce jeu en réutilisant au maximum les fonctions déjà implémentées. Vous nous décrirez les règles choisies et illustrerez par des exemples votre nouveau jeu.*

Bonus 3 : Intelligence Artificielle (3 points)

Exercice 7 *Le dernier point est de définir un ou plusieurs niveaux d'intelligence artificielle, afin de joueur contre la machine. Nous vous laissons libre de développer l'IA pour le jeu de votre choix, ou bien d'en réaliser une générique pour l'ensemble des jeux possibles.*

6. Il est possible de demander à l'utilisateur de rentrer la nouvelle table complète pour jouer son coup.