Algorithmique et Programmation Fonctionnelle

Projet : Variations sur le rami (Compléments)

Indications

Afin de vous faciliter la tâche et de clarifier certains points concernant le projet, nous vous donnons les indications suivantes.

1 Module Dictionnaire

Pour la fonction `remove`, il suffit de modifier le booléen associé au nœud correspondant au mot à enlever, il n'est pas nécessaire d'élaguer l'arbre résultant.

Le fichier dictionnaire suggéré contient des caractères qui, bien que parties intégrantes de notre belle langue et indispensables à l'orthographe correcte des mots, sont néanmoins une plaie à manipuler en informatique surtout lorsque l'on a décidé que l'alphabet comporte 26 lettres.

Le fichier `complements.ml` vous propose deux fonctions qui éliminent les mots contenant des caractères autres que les 26 imposés.

Point de détail : ce code insère des mots de longueur < 3 qui sont interdits par la règle, mais ce test est traité ailleurs (il pourrait l'être dès le dictionnaire).

2 Tirage au sort dans un multi-ensemble

Lorsqu'un joueur pioche une tuile, ou lors de la distribution initiale, il est nécessaire de pouvoir choisir une tuile de la pioche. Pour ne pas jouer tout le temps la même partie, on aimerait que ce choix soit aléatoire et change d'une exécution du programme à l'autre. Nous utiliserons pour cela le module `Random`.

La fonction `Random.int` de type `int` $\rightarrow$ `int` retourne un entier au hasard compris entre 0 inclus et son argument exclus ; par exemple `Random.int 42` est un entier x avec entre $0 \leqslant x \leqslant 41$.

Afin de pouvoir calculer des entiers qui ont l'air aléatoires, le module `Random` a besoin d'être initialisé (avant de tirer des valeurs) à l'aide d'une valeur initiale qui déterminera la suite des entiers tirés. OCaml dispose pour cela de la fonction `Random.self_init` de type `unit` $\rightarrow$ `unit` qu'il faudra appeler une fois au début de votre programme et qui initialise le générateur de nombres aléatoires avec une graine qu'il choisit automatiquement.

3 Interaction avec l'utilisateur

Il vous est demandé d'écrire une fonction `lit_coup` de type

```
string → Rule.main → Rule.combi list → bool →
    (Rule.main * (Rule.combi list)) option.
```

Le premier argument est le nom du joueur dont c'est le tour, il ne sert qu'à indiquer aux 17 personnes devant l'écran qui doit jouer à ce moment là. Le deuxième argument est la main du joueur au début de son tour, le troisième est l'état du jeu (ce qui est posé) au début de son tour, et le quatrième argument est un booléen indiquant si le joueur a posé (ou non), car la validation d'un coup a besoin de connaitre cette information.

La valeur de retour est une `option` qui vaudra `None` pour indiquer que le joueur souhaite passer. Sinon, il s'agira d'un couple constitué de la nouvelle valeur de la main du joueur, et

— soit du nouvel état du jeu (tout ce qui est posé) dans le cas où le joueur avait déjà posé ;

— soit des combinaisons que le joueur a posé en propre lors de son premier tour. Nous supposons en effet pour simplifier les choses que lors de son premier tour un joueur n'a pas accès aux combinaisons déjà posées par les autres joueurs.

La même convention est utilisée dans la fonction `coup_valide` : l'argument indiquant le nouveau jeu correspond soit à la table complète (cas où le joueur a déjà posé), soit uniquement aux combinaisons posées par le joueur (cas de la première pose du joueur).

Pour demander à l'utilisateur ce qu'il souhaite jouer, le plus simple à écrire est peut-être de lui demander de taper la totalité de la table après son coup. Cela demande beaucoup de copier-coller lorsque l'on joue, mais cela simplifie grandement l'interaction du point de vue du programmeur.

4 Grammaire pour la lecture d'un jeu

Pour l'analyse syntaxique, on donne la grammaire suivante :

$$
\begin{array}{rcl}
S & ::= & \texttt{(}\ \texttt{joueurs}\ Jl\ \texttt{)}\ \texttt{(}\ \texttt{jeu}\ Cl\ \texttt{)}\ \texttt{(}\ \texttt{pioche}\ Tl\ \texttt{)}\ \texttt{(}\ \texttt{tour int}\ \texttt{)} \\
Jl & ::= & J \mid J\ Jl \\
J & ::= & \texttt{(}\ \texttt{ident int}\ B\ C\ \texttt{)} \\
B & ::= & \texttt{true} \mid \texttt{false} \\
C & ::= & \texttt{(}\ Tl\ \texttt{)} \\
Cl & ::= & C\ Cl \mid \varepsilon \\
Tl & ::= & T\ Tl \mid \varepsilon
\end{array}
$$

Ici `int` désigne une nombre tandis que `indent` désigne un « identifiant », c'est-à-dire une succession de caractères non blancs quelconques. Les lexèmes `LPar` et `RPar` ont été remplacés par `(` et `)` respectivement pour plus de lisibilité.

Le non-terminal S désigne l'état complet du jeu (donc ce que votre analyse syntaxique doit parser), tandis que le non-terminal T correspond à la représentation d'une tuile qui dépend forcément de la règle du jeu considérée. Par exemple `"A"` est une représentation d'une tuile dans le rami des lettres, tandis que `"(rouge 4 )"` est une représentation possible de la tuile rouge de valeur 4 dans le Rummikub (au niveau lexical). Comment accomoder des représentations différentes des tuiles en fonction du jeu ? C'est l'objet de la fonction `lit_valeur` qui prend en argument une liste de `token` pour produire une valeur. Par exemple pour le rami des lettres on a :

```
lit_valeur [TGen "A"] = 'A'
```

si on a choisi de représenter les valeurs du rami des lettres par des `char`[1], et pour le Rummikub

```
lit_valeur [LPar; TGen "rouge"; TGen "4"; RPar] = Rouge 4
```

avec un choix approprié de `Rummikub.t`.

Dernière question : comment est-ce que l'analyseur syntaxique du foncteur `Jeu` sait à quel endroit couper la liste de lexèmes pour appeler la fonction `lit_valeur` ? La règle est la suivante : une valeur de tuile T est lue par le plus petit préfixe de lexèmes bien parenthésés.

Par exemple en parsant dans le flux de lexèmes le sous-flux suivant :

```
[< LPar; TGen "rouge"; TGen "4"; RPar; TGen "joker" >]
```

il faudra appeler la fonction `lit_valeur` avec commes paramètres `[ LPar; TGen "rouge"; TGen "4"; RPar]` puis `[TGen "joker"]`.

1. `type t = char` dans le module.