

TP 2

0 Environnement de travail

Les TP se font sous Linux. Si votre compte n'est pas encore activé (login : `prenom.nom`, mot de passe : celui de votre email `@universite-paris-saclay.fr`), vous pouvez utiliser pour les premières séances la session de secours, dont les logins et mot de passes sont affichés dans la salle.

0.1 Configuration de l'environnement

Les instructions de cette section de doivent être effectuée **qu'une seule fois, lors de la première séance** (sauf si vous utilisez la session de secours, il faudra alors le refaire sur votre vraie session lorsqu'elle sera activée).

1. ouvrir un terminal
2. ouvrir le fichier `~/.bashrc` au moyen d'un éditeur de texte
`$ gedit ~/.bashrc`
3. ajouter la ligne `source /public/kn/setup_ocaml.sh` en fin de fichier

Attention vérifier que la ligne n'est pas déjà présente dans le fichier, ce qui est possible si vous avez suivi l'UE IPF de L2 par exemple. Dans cas, ne pas ajouter 2 fois cette ligne. Vous devez ensuite **refermer tous les terminaux** puis en rouvrir un.

0.2 Utilisation de Visual Studio Code

Une fois configuré correctement (par le script d'initialisation), VSCode est un éditeur confortable pour du code OCaml (moins lourd que Netbeans ou Eclipse en particulier). Voici l'ensemble minimal des commandes pour les TP :

- Création d'un nouveau Fichier : **CTRL-N** (ou « *File* → *New file* ». Attention, le fichier nouvellement créé n'a pas de nom. Il convient alors de l'enregistrer (**CTRL-S**) ou *Save...*) en lui donnant un nom se terminant par `.ml`
- Évaluation d'une expression OCaml : il est possible de surligner (**SHIFT+↑ / ↓** ou en utilisant la souris) une portion de code OCaml puis de l'envoyer dans un interpréteur avec **SHIFT-ENTER**. Attention, il convient d'évaluer les définitions dans l'ordre.

De façon générale, les archives de TP contiennent déjà un ou plusieurs fichier à compléter, il est rare de devoir créer des fichiers.

1 Le Solitaire

Le but du TP est de coder des fonctions utilitaires permettant de jouer à une variante du Solitaire. Ce jeu (aussi appelé *Klondike* en anglais) est un jeu utilisant 52 cartes disposées de façon particulière. Quatre zones, initialement vides sont réservées pour les cœurs, piques, trèfles et carreaux. À côté d'eux, la défausse (initialement vide) et la pioche consistent en deux tas de cartes. La cartes de la pioche sont face cachée.

Enfin, 7 piles de cartes sont disposées sous les zones de rangement. Initialement les tas ont respectivement 0, 1, 2, ..., 6 cartes cachées et 1 carte visible. La figure 1.a montre une configuration initiale de notre jeu. Dans cette figure, les carrés pointillés représentent des emplacements vides (sans carte). De plus, afin de simplifier les fonctions d'affichage, les piles de cartes sont affichées horizontalement et précédées de leur numéro (plutôt que verticalement). Le but du jeu est de ranger toutes les cartes, dans l'ordre sur chacun des quatre emplacements réservés pour chaque couleur (une couleur ici est cœur, pique, trèfle ou carreau, pas noir ou rouge). Les règles sont très simples.

On dit qu'une carte est *directement accessible* si elle est soit sur la défausse, soit le plus à gauche de l'une des piles. Dans la figure 1.f, les cartes directement accessibles sont : le 9 de trèfle (sur la défausse), le Roi de pique, Valet de trèfle, 3 de carreau, 3 de pique, Roi de trèfle, 8 de carreau et 9 de carreau.

Mouvements des cartes :

- on peut déplacer un As sur la zone de rangement vide de sa couleur
- si une carte est *directement accessible* et qu'elle est immédiatement supérieure à la carte au sommet de la zone de rangement de sa couleur, alors on peut l'y ranger (i.e. si la dernière carte posée sur la zone des cœurs est le 10 et que le Valet de cœur est accessible, on peut le ranger au dessus du 10. Par contre si la dernière carte posée sur la zone des cœurs est le 8 et que le Valet de cœur est accessible, on ne peut pas le ranger car il ne suit pas immédiatement le 8).
- on peut déplacer la carte au sommet de la zone de défausse pour la mettre sous une carte visible des piles numérotées, si elle est de valeur directement inférieure et de couleur alternante (i.e. on peut placer un 7 de pique sous un 8 de cœur ou un 8 de carreau)
- on peut déplacer toute une suite de cartes d'une pile à une autre, la carte la plus en dessous devant être directement inférieure et de couleur alterante à la carte de destination. Autrement dit, on peut déplacer une carte non directement accessible d'une pile si on déplace aussi toutes les cartes en dessous.
- enfin, si une des 7 piles est vide, on peut uniquement placer un Roi dessus.

À partir de la figure 1.a, on peut par exemple *ranger* l'As de cœur qui est directement accessible sur la pile numéro 5, ce qui donne la configuration de la figure 1.b, dans laquelle la carte est rangée et le 9 de carreau qui se trouvait en dessous est dévoilé.

Ensuite, on peut choisir de déplacer le 3 de carreau (pile 2) sous le 4 de pique (pile 4). On obtient la figure 1.c. Le Valet de trèfle devient ainsi visible. On peut ensuite déplacer le 3 de carreau **et** le 4 de pique sous le 5 de cœur (pile 3). Le résultat de cette opération est donné à la figure 1.d.

On peut ensuite déplacer le 9 de carreau (pile 5) sous le 10 de pique (pile 7), comme dans la figure 1.e.

À ce stade, on se retrouve bloqué, on va donc choisir de piocher (il est toujours possible de piocher même quand on peut jouer autre chose), ce qui révèle 9 de trèfle dans la zone de défausse.

On procède ainsi, en piochant lorsque l'on est bloqué jusqu'à avoir réussi à ranger toutes les cartes (ou à abandonner car on se retrouve complètement bloqué). Si à un moment la pioche est vide, on retourne la défausse pour en faire une nouvelle pioche.

Une fois le programme lancé, la partie commence. Les commandes possibles sont :

- P pour piocher une carte ;
- Ri pour ranger la carte se trouvant en i . Ce dernier peut être soit la lettre D pour indiquer que l'on veut ranger la carte se trouvant dans la défausse sur son emplacement finale, soit un entier entre 1 et 7 pour indiquer que l'on veut ranger la carte se trouvant le plus à gauche de la pile indiquée
- Di j pour déplacer le plus de cartes possible entre i et j . Ici, i est soit D soit un entier de 1 à 7 et j est un entier de 1 à 7.
- Q pour quitter.

Ainsi, pour obtenir les figures 1.a 1.f, on entre successivement les commandes :

- R5 (on range l'As se trouvant dans la pile 5)
- D24 (on déplace le 3 de la pile 2 sous le 4 de la pile 4)
- D43 (on déplace le 3 et 4 de la pile 4 sous le 5 de la pile 3)
- D57 (on déplace le 9 de la pile 5 sous le 10 de la pile 7)
- P (on pioche)

2 Structure du programme et méthode de travail

2.1 L'archive Zip

L'archive contient les fichiers suivants :

solitaire_defs.ml Un fichier contenant uniquement la définition de types OCaml pour représenter les cartes et la configuration du jeu, ainsi qu'une fonction et plusieurs variables globales. utilitaires. Il convient de lire attentivement les définitions de types, **mais il ne faut absolument pas modifier ce fichier.**

```

1  type valeur = Valeur of int | Valet | Dame | Roi
2
3  type couleur = Coeur | Pique | Carreau | Trefle
4
5  type carte = { couleur : couleur; valeur : valeur }
6
7
8  type jeu = { coeur : carte list;
9              pique : carte list;
10             carreau : carte list;
11             trefle : carte list;
12             piles : (carte list * carte list) list;
13             defausse : carte list;
14             pioche : carte list }

```

Les cartes sont représentées comme lors des exemples du cours : une carte est un produit nommé à deux composantes (la valeur et la couleur). Le type **valeur** est un type sommes contenant trois constantes (**Valet**, **Dame**, **Roi**) et un constructeur avec argument représentant les cartes numériques. Le type **couleur** est composé de quatre constantes.

Dans la suite on appellera un *tas* de carte une simple liste de cartes. Une *pile de cartes* est par contre représenté par une paire de listes de cartes : les cartes visibles et les cartes cachées. Le type **jeu** est un produit nommé qui représente une configuration de jeu. Les champs **coeur**, **pique**, **carreau**, **trefle** contiennent des tas de cartes (**carte list**) qui sont les zones de rangement. Les champs **defausse** et **pioche** représentent ces deux tas du jeu. Enfin, le champ **piles** représente les 7 piles de cartes. Le type de ce champs est **(carte list * carte list) list**, c'est donc une liste de paires. Cette liste sera de taille 7 en pratique.

Enfin, le fichier définit une fonction :

```

32  let string_of_carte c = ...

```

Cette fonction, du type **carte -> string** renvoie la chaîne de caractère permettant d'afficher la carte convenablement, avec les couleurs. Par exemple, effectuer

```
Printf.printf "%s\n"(string_of_carte { valeur=Roi; couleur=Coeur })
```

affichera :

 K♥

Le fichier contient enfin deux variables globales :

```

49  let carte_cachee = ...
50
51  let zone_vide = ...

```

La première est une chaîne de caractères correspondant à «  » représentant une carte retournée et la seconde contient une chaîne correspondant à « ::::: » représentant un tas de cartes vide.

solitaire.ml : le fichier que vous devez compléter contenant la logique du jeu (détection déplacement de cartes, affichage du jeu).

main.ml : le programme principal qui utilise votre fichier **solitaire.ml**. Il gère les saisies de l'utilisateur, l'initialisation, Vous pouvez le lire, mais ne devez pas le modifier.

dune, **dune-project** : un fichier de configuration pour la commande **dune** qui va compiler votre programme.

2.2 Compilation et tests

Pour compiler votre programme, vous ne devez pas appeler **ocamlc** directement mais utiliser la commande **dune build** dans le répertoire où se trouve votre code :

- la commande **dune build** écrite seule dans le terminal, compile votre fichier **solitaire.ml** et produit deux fichiers exécutables **main.exe** et **test.exe**.

Une fois toutes les fonctions complétées, vous pourrez tester votre jeu en lançant (`./main.exe`). Lorsque vous lancez ce programme, le choix des cartes est aléatoire, ce qui peut compliquer les tests. Si vous les lancez avec l'option `-norandom` comme ceci :

```
./main.exe -norandom
```

La partie jouée sera alors toujours la même, ce qui peut faciliter le débogage.

- l'exécutable `test.exe` est fourni pour vous permettre de tester vos fonctions une à une à l'aide de tests unitaires. Vous pouvez le compléter comme vous souhaitez puis le lancer avec `./test.exe` (le programme `main.exe` ne fonctionnera que si toutes les fonctions sont complétées).

2.3 Utilisation du corrigé

Afin que le TP ne soit pas un exercice à tiroir (où rater la première question implique de rater la suite), un fichier binaire `help_solitaire.cmo` est fourni. Il contient le corrigé de toutes les fonctions. Vous ne pouvez évidemment pas retrouver le code source correspondant, mais vous pouvez l'utiliser. Le début du fichier `solitaire.ml` à compléter est comme suit :

```
1           la fonction string_of_carte (qui s'occupe d'ajouter les couleurs correct
2 *)
3
4 let affiche_tas tas =
5     failwith "TODO Q1"
6
7 (* Q2
8     affiche_piles : (carte liste * carte list) list -> unit
9     Affiche la liste des piles. Chaque pile est précédée de son numéro, en commençant
10    Pour chaque pile (lvisibles, lcachees) on affiche :
11    - son numéro, suivi de deux points ":"
12    - les cartes de lvisibles, chacune suivie d'un espace. Comme précédemment les car
convertir avec
13        string_of_carte.
14    - pour chaque carte de lcachees, on affiche la chaine contenue dans la variable g
15    - deux retour à la ligne
16
17    Une fonction utile est la fonction List.iteri: (int -> 'a -> unit) -> 'a list -> unit
```

À la ligne 1, l'instruction `open Solitaire_defs` permet d'inclure les définitions du fichier `solitaire_defs.ml` à cet endroit. Vous pouvez donc considérer que les types `jeu`, `carte`, `valeur` ainsi que toutes les variables du fichier sont visibles ici.

La fonction `affiche_tas` (ainsi que toutes les suivantes) appelle directement la fonction du corrigé du même nom. Vous devez évidemment remplacer le corps de la fonction par votre propre code, mais, si vous êtes bloqué pour une fonction vous pouvez

- mettre votre code en commentaire. Il sera tout de même corrigé.
- revenir au code initial qui appelle le corrigé.

Cela vous permettra de ne pas rester bloqué sur une question et d'avancer aux questions suivantes sans être pénalisés pour les suivantes.

Attention, ceci ne fonctionne que sur les machines du PUIO. Pour les étudiants utilisant leur machine personnelle, vous pouvez télécharger l'archive `_ext.zip` sur la page du cours, et vous serez en mode un peu dégradé (le corrigé pré-compilé n'est pas disponible).