

Devoir à la maison

1 Modalités

Le devoir est **individuel**. Il est à rendre au plus tard :

le lundi 10 janvier 2021, 17h00

Le seul fichier à rendre est le fichier **solitaire.ml** que vous aurez complété avec votre solution. Les autres fichiers ne doivent être ni modifiés ni rendus. Le dépôt se fera uniquement sur l'espace eCampus du cours, dans l'activité de rendu prévue à cet effet. Le devoir peut être déposé sous forme de « brouillon », mais le dépôt final devra être validé avant la date limite. Attention, **une fois validé**, votre soumission ne pourra plus être modifiée.

Le plagiat sera sévèrement sanctionné, conformément au règlement des études premier cycle de l'Université. Les questions génériques (de compréhension de l'énoncé, ou de problème pour utiliser le code) peuvent être posée sur le forum du cours, sur la page eCampus (afin que les réponses puissent profiter à tout le monde).

Les questions personnelles sur votre devoir, en particulier montrant des exemples de votre code, doivent être posée individuellement par email (kim.nguyen@universite-paris-saclay.fr) afin de ne pas influencer les autres étudiants.

La suite de l'énoncé se décompose en trois parties. La section 2 présente le cadre général du sujet (le jeu du Solitaire) et les règles du jeu. La section 3 explique la structure des fichiers fournis et vous donne toutes les commandes pour tester votre programme facilement et le comparer à la solution donnée dans l'archive. Enfin, la section 4 contient les questions.

Le projet a été conçu pour être fait sous l'interface Jupyterhub disponible en ligne (en particulier, le programme de référence et le corrigé ont été compilés sous cet environnement). Si, vous ne voulez vraiment pas utiliser l'environnement JupyterHub, vous pouvez me contacter pour obtenir une version compilée du corrigé pour votre plateforme de travail. Une autre solution est de développer dans votre environnement favori et de n'utiliser JupyterHub que pour tester le corrigé.

2 Le Solitaire

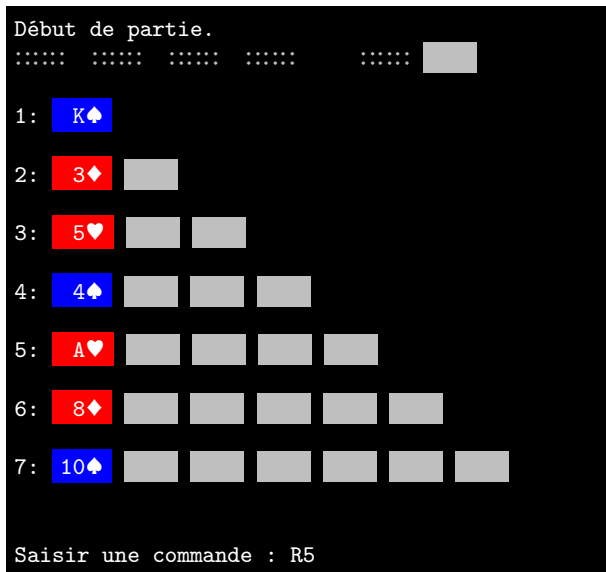
Le but du devoir est de coder des fonctions utilitaires permettant de jouer à une variante du Solitaire. Ce jeu (aussi appelé *Klondike* en anglais) est un jeu utilisant 52 cartes disposées de façon particulière. Quatre zones, initialement vides sont réservées pour les cœurs, piques, trèfles et carreaux.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

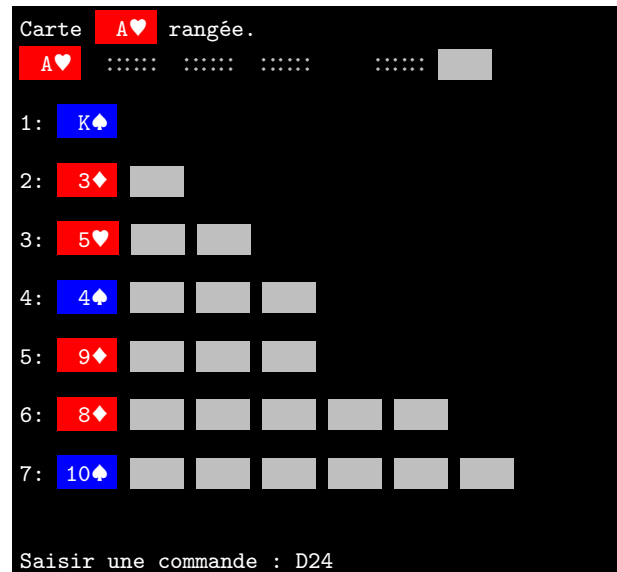
On dit qu'une carte est *directement accessible* si elle est soit sur la défausse, soit le plus à gauche de l'une des piles. Dans la figure 1.f, les cartes directement accessibles sont : le 9 de trèfle (sur la défausse), le Roi de pique, Valet de trèfle, 3 de carreau, 3 de pique, Roi de trèfle, 8 de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

Mouvements des cartes :

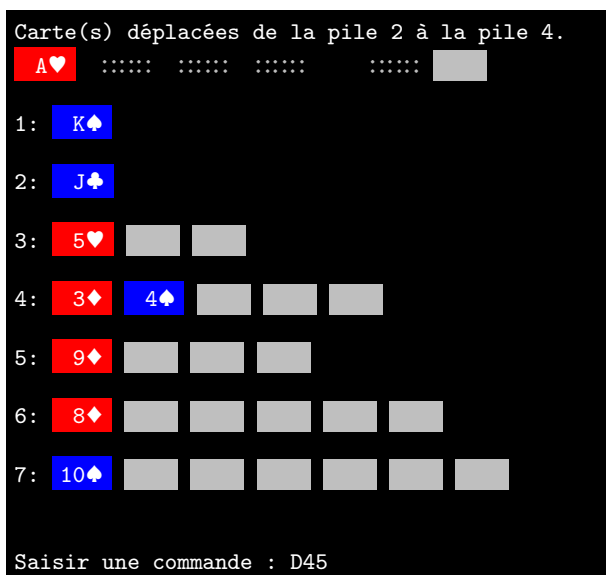
— on peut déplacer un As sur la zone de rangement vide de sa couleur



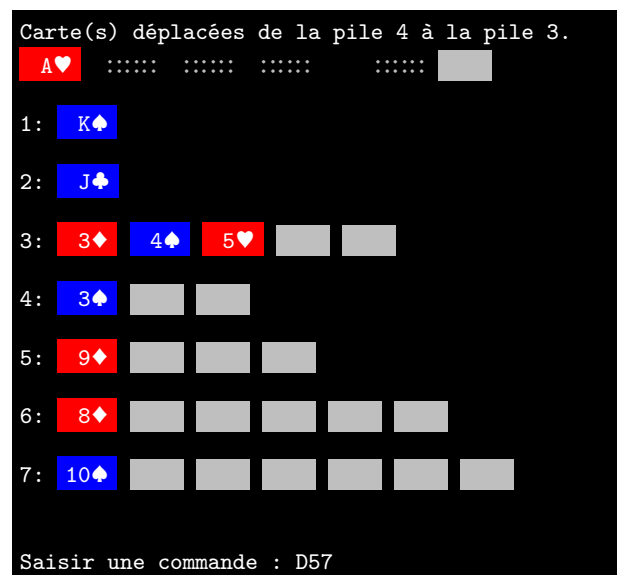
(a) Une configuration initiale.



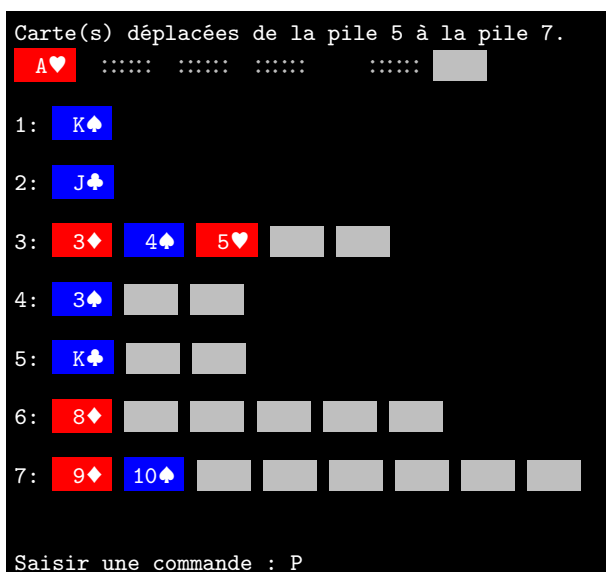
(b) On a rangé l'As de cœur.



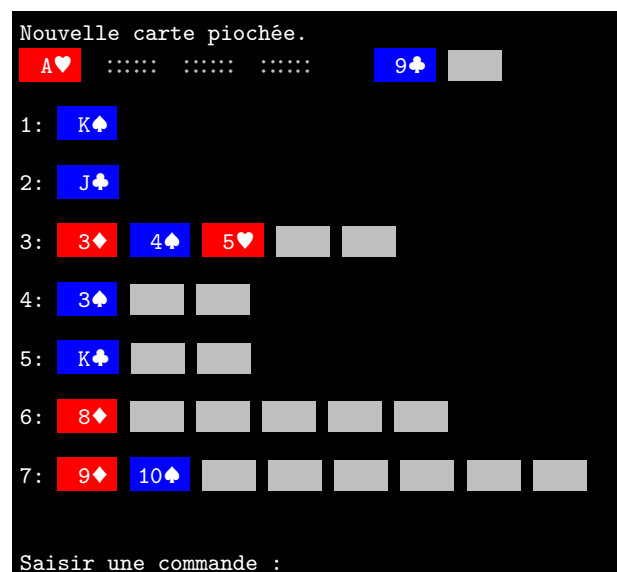
(c) On a déplacé le 3 de carreau.



(d) On a déplacé le 3 et le 4 sous le 5.



(e) On a déplacé le 9 sous le 10.



(f) On a pioché.

FIGURE 1 – Plusieurs étapes du solitaire

- 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.

Le programme fourni en exemple se lancer simplement avec la commande :

```
./solitaire_ref.exe
```

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 à 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)

3 Structure du programme et méthode de travail

Nous donnons ici quelques indications sur la façon de travailler en particulier comment lire les fichiers donnés et travailler avec OCaml sur JupyterHub.

3.1 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_defs.cmi, **solitaire_defs.cmo**, **help_solitaire.cmi** et **help_solitaire.cmo** : des fichiers compilés (binaires) contenant le code du corrigé. Il ne faut ni effacer ni modifier ces fichiers. Si cela vous arrive, il faudra les récupérer depuis l'archive originale. Attention, si vous décompressez de nouveau l'archive, les fichiers existants seront écrasés. Il convient donc de faire une copie de votre fichier **solitaire.ml** contenant vos réponses avant de décompresser de nouveau l'archive, sinon vous risquez de tout perdre!

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.

Makefile : un fichier de configuration pour la commande **make** qui va compiler votre programme.

3.2 Compilation et tests

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

- la commande `make` écrite seule dans le terminal, compile votre fichier `solitaire.ml` et produit un fichier exécutable `solitaire.exe`. Vous pouvez tester votre programme en le lançant (`./solitaire.exe`) et le comparer au programme de référence (`./solitaire_ref.exe`).

Lorsque vous lancez ces programmes, le choix des cartes est aléatoire, ce qui peut compliquer les tests. Si vous les lancer avec l'option `-norandom` comme ceci :

```
./solitaire_ref.exe -norandom
```

alors le déroulement de la partie sera le même que celui indiqué à la figure 1 à chaque fois, ce qui peut vous permettre de reproduire une même partie entre le programme de référence et le votre.

- la commande `make test` lance un interpréteur OCaml avec les fonctions de votre fichier `solitaire.ml` chargées à l'intérieur. Vous pouvez donc directement appeler vos fonctions pour les tester. Par exemple, pour tester la fonction `lance_de : unit -> int` de la question 1 :

```
make test
```

```
OCaml version 4.08.1
```

```
# init_jeu;;
- : unit -> Solitaire_defs.jeu = <fun>
# let jeu = init_jeu ();;
val jeu : Solitaire_defs.jeu = ...
# affiche_jeu jeu;;
...
```

Le programme compile votre fichier et lance l'interpréteur OCaml (ces commandes sont affichées dans le terminal). Vous pouvez ensuite saisir des expressions OCaml. Vous pouvez quitter l'interpréteur avec CTRL-D.

3.3 Utilisation du corrigé

Afin que le devoir 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 open Solitaire_defs
2 (* Q1
3     affiche_tas : carte list -> unit
4     Affiche
5     — soit un emplacement vide (en utilisant la variable globale zone_vide)
6     si le tas est vide
7     — soit la première carte du tas s'il est non vide. On peut convertir la carte
8     en chaîne avec la fonction string_of_carte (qui s'occupe d'ajouter
9     les couleurs correctement).
10
11     0.75 point
12 *)
13
14 let affiche_tas tas =
15 (* Commentez la ligne ci-dessous et mettez votre code.
16    Si votre code ne fonctionne pas, commentez le et remettez cette ligne. *)
17
18    Help_solitaire.affiche_tas tas
```

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 bloquer sur une question et d'avancer aux questions suivantes sans être pénalisés pour les suivantes (évidemment si vous ne répondez pas du tout à une question et laissez le code initial, vous aurez 0 point).

3.4 Travail sous JupyterHub

Il est conseillé de travailler dans l'environnement JupyterHub. Ce dernier est accessible à l'URL suivante :

<https://jupyterhub.ijclab.in2p3.fr/>

Sur cette URL cliquer sur « Se connecter ». Choisir Université Paris-Saclay comme fournisseur d'identité, puis se connecter.

Si le site affiche un gros bouton bleu « Start My Server », cliquer dessus. L'instance JupyterHub démarre alors. Il est suggéré de travailler de la façon suivante :

1. la première fois, cliquer sur le bouton **Upload** (en haut à droite) et déposer l'archive Zip contenant le devoir.
2. aller dans le menu **New** puis cliquer **Terminal**. Un terminal se lance dans un nouvel onglet.
3. aller dans le terminal nouvellement ouvert et exécuter la commande :

```
unzip devoir_l2_info_2021-2022.zip
```

Ce qui aura pour effet de décompresser l'archive.

À partir de maintenant vous pouvez travailler. À chaque fois que vous voulez avancer sur le devoir

1. se connecter à JupyterHub
2. se déplacer dans le répertoire `devoir_l2_info_2021-2022`
3. ouvrir un terminal (Menu **New**, **Terminal**)
4. dans le terminal se placer dans le bon répertoire

```
cd devoir_l2_info_2021-2022
```
5. cliquer dans l'explorateur de fichier sur le fichier `solitaire.ml` (et les autres fichiers si vous voulez les lire)
6. éditer le fichier `solitaire.ml`
7. basculer dans le terminal et faire `make` pour compiler. En cas d'erreur, retourner dans le fichier pour corriger
8. tester vos fonctions, soit en lançant le programme `solitaire.exe` soit en faisant `make test` et en les testant dans l'interpréteur OCaml.

Vous pourrez progresser en répétant les étapes 6, 7 et 8 pour chaque question.

Il faudra enfin télécharger le fichier `solitaire.ml` (le cocher dans l'explorateur JupyterHub) puis cliquer **Download**. Ce fichier devra ensuite être déposé, avant la date limite sur eCampus.

4 Questions

Les questions sont données en commentaires dans le fichier. Il y a 22 questions, de Q0 à Q21. Les questions peuvent étre de deux natures :

- donner le code d'une fonction (à la place de l'appel au corrigé)
- donner le type d'une fonction dont le code est donné

À titre d'indication, les questions de types et les « petites » fonctions (non récursives par exemple) font moins de 1 point. Les fonctions plus complexes font de 1 à 2 points. Il y a en gros 10 points sur des petites fonctions et 10 points sur les fonctions un peu plus complexes. Les seules fonctions véritablement complexes sont celles de la question Q19 (2 points). Il y a deux fonctions mais **vous pouvez choisir laquelle vous voulez compléter** et pouvez laisser l'autre par défaut.