TP 15 d'option informatique - MPSI

Une grille de sudoku est un tableau composé de 9 lignes, 9 colonnes et 9 carrés où certaines cases contiennent des chiffres. Le but du jeu est de compléter les 81 cases de telle manière que chaque chiffre entre 1 et 9 apparaisse exactement une fois par ligne, une fois par colonne et une fois par carré. Par exemple, voici une configuration initiale et sa solution :

	4					3		
			1		2			6
1		3			5	7		
	5	2	3		9		6	
	7		6		1	8	3	
		7	8			5		1
4			5		3			
		1					8	

6	4	5	9	8	7	3	1	2
7	8	9	1	3	2	4	5	6
1	2	3	4	6	5	7	9	8
8	5	2	3	4	9	1	6	7
3	1	6	7	5	8	9	2	4
9	7	4	6	2	1	8	3	5
2	3	7	8	9	6	5	4	1
4	6	8	5	1	3	2	7	9
5	9	1	2	7	4	6	8	3

1. Écrire une fonction pour résoudre une grille de sudoku. Pour réfléchir au problème, utilisez une feuille et organisez votre programme avec des fonctions intermédiaires. Si vous ne savez pas comment faire, vous pouvez lire les indications des parties suivantes.

   Vérifier que votre programme résout l'exemple ci-dessus et qu'il indique lorsqu'une grille n'a pas de solution.
2. Vérifier que si dans l'exemple ci-dessus, on supprime les chiffres des deux dernières lignes de la configuration initiale (le 4, le 5, le 3, le 1 et le 8) alors la grille possède 3012 solutions.

Bonus : afin de rendre votre programme plus rapide, vous pouvez implémenter d'autres techniques de résolution, voir par exemple `https://www.sudoku129.com/grilles/tips_intro.php`.

Indications (réfléchissez seul avant de lire ce qui suit)

Dans la stratégie décrite ici, on va faire un choix (c'est à dire imposer un certain chiffre pour une case) et si ce choix n'est pas bon, on revient en arrière. Cette technique s'appelle du ***backtracking*** ("retour sur trace"). L'idée est de créer pour chaque case de la grille une liste contenant les chiffres compatibles avec toutes les conditions sur les lignes, sur les colonnes et sur les carrés. Par exemple, dans l'exemple de l'énoncé, la case en bas à gauche ne peut pas contenir de 1, 4, 7 ou 8. La liste correspondante est donc `[2; 3; 5; 6; 9]`. Lorsque les 81 listes ont été créées, on a plusieurs cas :

- Toutes les cases contiennent un chiffre. Dans ce cas, la résolution est terminée.
- Si l'une des listes est vide, c'est que la case ne peut pas être complétée et la grille n'a pas de solution.
- Sinon, il existe une liste non vide. On choisit l'une de ces listes notée `li`, on choisit le premier élément de `li` noté `n` et on impose que la case contienne le chiffre `n`. On met alors à jour les listes et on recommence récursivement la résolution. Deux cas peuvent se présenter lors de l'appel récursif :
  - Une solution est trouvée, c'est que notre choix était le bon et la résolution est terminée.
  - Il n'y a pas de solution, on en déduit que la case ne peut pas contenir `n`. On recommence donc avec le deuxième chiffre de la liste `li`.

  Finalement, si tous les chiffres de `li` ont été essayés sans trouver de solution, c'est que le problème n'en a pas.

3. Répondre aux questions 1 et 2 avec ces indications. Si vous n'y arrivez pas, lisez ce qui suit.

Plus d'indications (réfléchissez seul avant de lire ce qui suit)

On définit les types suivants :

```
type cases = int array array;;
type grille = {
    cases: cases;
    c_poss: int list array array
  };;
```

```
type coup = {i: int; j: int; n: int};;
type tour = {
    c: coup;
    c_incomp: coup Stack.t;
  };;
```

On utilisera également le type `'a option` (il est déjà défini en OCaml, vous n'avez pas besoin de la redéfinir) :

```
type 'a option =
  | None
  | Some of 'a;;
```

Une matrice `mat` de type `cases` sera de taille 9×9 et va représenter la grille de sudoku. Ainsi, pour `i, j` $\in [\![0;8]\!]$, l'entier `mat.(i).(j)` vaut `0` si la case n'a pas été complétée et vaut `n` $\in [\![1;9]\!]$ lorsqu'elle contient `n`. Une variable `g` de type `grille` résume les informations utiles sur une grille de sudoku :

- Le champ `cases` donne accès aux cases qui ont été complétées (voir la description du type `cases` ci-dessus).
- Le champ `c_poss` donne accès aux coups possibles pour chaque case. Ainsi, `g.c_poss.(i).(j)` contient la liste dont on a parlé dans la partie précédente (celle qui contient tous les chiffres compatibles avec les conditions sur les lignes, sur les colonnes et sur les carrés).

La méthode de backtracking nécessite de pouvoir annuler des coups s'ils se sont avérés incorrects. Pour cela, on représente un coup par une variable `c` de type `coup`. Les entiers `c.i` et `c.j` sont la ligne et la colonne de la case jouée, et `c.n` est le chiffre qui a été placé dans la case. Par exemple, le fait de placer le chiffre `2` dans la case en haut à droite correspond à la variable `c = {i = 0; j = 8; n = 2}`.

Lorsqu'un coup est joué, certains éléments des listes de `g.c_poss` sont supprimés. Ces suppressions seront mémorisées dans une variable de type `tour`. Ainsi, pour chaque coup joué, on créé une variable `t` de type `tour` telle que `t.c` contient le coup joué et `t.c_incomp` est une pile contenant tous les coups supprimés de `g.c_poss` du fait de leur incompatibilité avec `t.c`. L'intérêt de la variable `t` est de pouvoir annuler le coup si celui-ci ne permet pas d'aboutir à une solution.

4. Écrire une fonction « `list_n_to_1: int -> int list` » qui prend en entrée un entier `n` et renvoie la liste `[n; n-1; ...; 1]`.
5. Écrire une fonction « `nv_coup_incomp : coup -> grille -> tour -> unit` » qui prend en entrée une variable « `c: coup` » (représentant un coup ne pouvant plus être joué) ainsi que « `g: grille` » et « `t: tour` ». Lorsque le coup `c` est présent dans `g.c_poss`, votre fonction doit le supprimer, puis l'ajouter à `t.c_incomp`. Lorsque le coup `c` n'est pas présent dans `g.c_poss`, votre fonction ne fait rien. On pourra supposer que `c` apparaît au plus une fois dans `g.c_poss`.
6. Écrire une fonction « `nouveau_coup : coup -> grille -> tour` » qui prend en entrée un coup « `c: coup` » que l'on veut jouer ainsi que « `g: grille` ». Votre fonction doit mettre à jour la case de `g.cases` correspondant à `c` et supprimer de `g.c_poss` tous les coups incompatibles avec `c`. De plus, votre fonction doit renvoyer un objet « `t: tour` » tel que `t.c` vaut `c` et `t.c_incomp` contient tous les coups qui ont été supprimés de `g.c_poss`.
7. À l'aide des questions précédentes, écrire une fonction « `obtenir_grille : cases -> grille option` » qui prend en entrée une configuration initiale « `c_ini: cases` » et renvoie « `Some g` » où « `g: grille` » est la grille dans laquelle tous les chiffres présents dans `c_ini` ont été placés. Si les chiffres de `c_ini` sont incompatibles entre eux, votre fonction doit renvoyer `None`.

Tant que la grille n'est pas complétée, on sélectionne une case dont la valeur n'est pas connue pour en chercher la valeur. Étant donnée « `g: grille` », on choisit de compléter la case de coordonnée `(i,j)` vérifiant `g.cases.(i).(j) = 0` et telle que la liste `g.c_poss.(i).(j)` est de taille minimale.

8. Écrire une fonction « `trouver_prochaine_case: grille -> (int * int) option` » qui prend en entrée « `g: grille` » et renvoie « `Some (i,j)` » où « `(i,j): int * int` » sont les coordonnées de la prochaine case dont on va chercher la valeur. Votre fonction renverra `None` si tous les éléments de `g.cases` sont non nuls.
9. Écrire une fonction « `annuler_tour : grille -> tour -> unit` » qui prend en entrée deux variables « `g: grille` » et « `t: tour` », et qui annule le tour `t`. En d'autres termes, `g` doit être modifiée pour que l'élément de `g.cases` correspondant à `t.c` prenne la valeur `0` et que tous les coups de `t.c_incomp` soient ajoutés à `grille.c_poss`.
10. Écrire une fonction « `solve : cases -> cases option` » qui prend en entrée une grille initiale et renvoie la grille complétée. Dans le cas où il n'y a pas de solution, la fonction doit renvoyer `None`. Testez avec l'exemple donné en début de TP (voir la question 1).

Jusqu'à présent, une fois le sudoku résolu, on renvoyait la solution. Pour compter le nombre total de solutions possibles, lorsqu'une solution est trouvée, on incrémente un compteur et on continue la résolution.

11. Écrire une fonction « `nb_sols : cases -> int` » qui compte le nombre de solutions pour une grille de sudoku. Testez sur l'exemple donné au début du TP (voir la question 2).