1 Problème 1

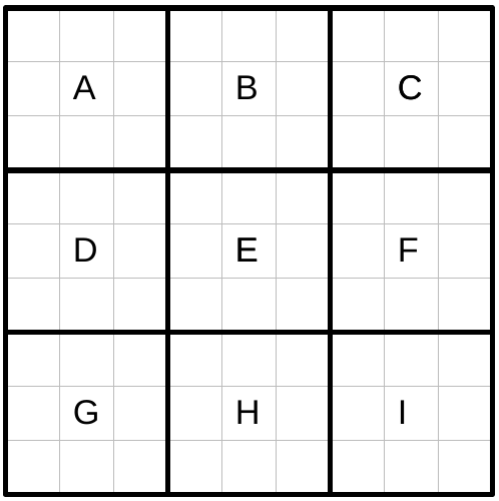


FIGURE 1 – La grille du sudoku, avec les 9 sous-grilles mises en évidence

```
[|[|Pleine 3; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Vide; Vide; Pleine 7; Vide; Vide; Vide|];
  [|Vide; Pleine 8; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Pleine 5|];
  [|Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide; Vide|];
  [|Vide; Vide; Vide; Pleine 2; Vide; Vide; Vide; Vide; Vide|];
  |]
```

FIGURE 2 – Grille de sudoku avec 5 valeurs initiales en Ocaml

On propose un type énumération pour représenter les cases d'une grille de sudoku :

```
type contenu = Vide | Pleine of int;;
```

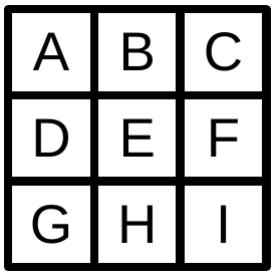
On représentera une grille de sudoku par le type `sudoku = contenu array array`.

1. Écrire une fonction `init : int*int*int list -> sudoku` qui crée une grille à partir d'une liste l de triplets (i, j, v), indiquant que la grille contient initialement la valeur v à la case ligne i, colonne j.
   Par exemple pour la liste `[(0,0,3); (3,5,7); (4,1,8); (8,3,2); (6,8,5)]` la fonction créera la grille présentée dans la Figure 2.

```
let init l =
  let g = Array.make_matrix 9 9 Vide in
  let rec aux l = match l with
    |[] -> ()
    |(i,j,v)::q -> g.(i).(j) <- Pleine v; aux q
  in
  aux l; g;;
```

On veut pouvoir vérifier si changer une case rend une grille invalide au vu des règles du sudoku. On va d'abord se concentrer sur la troisième contrainte.

Pour facilement énumérer les cases d'un sous-grille, on va attribuer des coordonnées aux-sous-grilles. Ces coordonnées sont attribuées en "oubliant" les cases du sudoku et en ne gardant que les sous grilles :



Les coordonnées de A sont (0,0)
Les coordonnées de B sont (0,1)
Les coordonnées de D sont (1,0)
Les coordonnées de F sont (1,2)
Les coordonnées de H sont (2,1)

FIGURE 3 – La grille des sous-grilles

Dans la suite on notera (a, b) les coordonnées des sous-grilles et (i, j) les coordonnées des cases du sudoku.

2. On considère une sous-grille telle que $b = 1$ (donc B, E ou H). Que peut on dire des valeurs possibles pour le j des cases contenues dans cette sous-grille ? Si on considère maintenant b quelconque, quelles sont les valeurs possibles pour j en fonction de b ?
   Les j possibles si $b = 1$ sont $3, 4, 5$, soit $3 * 1 + 0$, $3 * 1 + 1$ et $3 * 1 + 2$.
   En général, les j possibles sont $3 * b + 0$, $3 * b + 1$ et $3 * b + 2$.

3. Quelles sont les valeurs possibles pour i, en fonction de a ? Même chose, les i possibles sont $3 * a + 0$, $3 * a + 1$ et $3 * a + 2$.

Si on raisonne dans l'autre sens et qu'on veut déterminer les coordonnées (a, b) de la sous-grille à partir de celles (i, j) d'une case, on peut montrer qu'on a $a = (i/3) * 3$ et $b = (j/3) * 3$, où la division est **euclidienne**.

4. À l'aide des questions précédentes, écrire une fonction `troisieme_contrainte : grille -> int -> int -> int -> bool` qui prend en entrée une grille, un i, un j et une valeur n et vérifie qu'aucune case de la sous-grille de la case (i, j) ne contient déjà la valeur n.

```
let troisieme_contrainte g i j n =
  let a = (i/3) and b = (j/3) in (*coordonnées de la sous-grille*)
  let res = ref true in

  (*On parcourt toutes les case de la sous grille*)
  for ii=0 to 2 do
    for jj=0 to 2 do
      if g.(a*3+ii).(3*b+jj) = Pleine n then res:= false
    done;
  done;
  !res;;
```

5. Écrire une fonction `possible : grille -> int -> int -> int -> bool` telle que `possible grille i j n` renvoie un booléen indiquant s'il est licite de placer la valeur n à la ligne i et à la colonne j de la grille.

```
let possible g i j n =
  let res = ref (troisieme_contrainte g i j n) in (*On vérifie la 3e contrainte*)

  (*On parcourt toutes les case de la ligne i*)
  for jj = 0 to 8 do
    if jj <> j && g.(i).(jj) = Pleine n then res:=false
  done;

  (*On parcourt toutes les case de la colonne j*)
  for ii = 0 to 8 do
    if ii <> i && g.(ii).(j) = Pleine n then res:=false
  done;

  (*On a vérifié les trois contraintes*)
  !res;;
```

On suppose donnée une fonction `imprime : sudoku -> unit` qui affiche une grille proprement à l'écran. Notre but est maintenant d'écrire un programme affichant une grille solution.

6. Étant donné une grille solution partielle, remplie jusqu'à la case (i, j) qui n'est pas la dernière, comment construire une solution partielle remplie jusqu'à la prochaine case ? Il faudra commencer par déterminer les coordonnées de la prochaine case (attention au bouts de ligne).

   Si on parcourt les cases à partir de $(0, 0)$ de gauche à droite de droite à gauche, la prochaine case est $(i, j + 1)$ si $j \neq 8$ et $(i + 1, 0)$ si $j = 8$.

   Pour savoir quelle valeur peut aller dans la case suivante, on les teste toutes. Pour chaque valeur, on utilise la fonction `possible` pour vérifier si ça vaut la peine de continuer ou pas. Si oui, on remplit la case d'encore après. Sinon, on teste la valeur suivante. C'est le principe du retour sur trace.

7. En complétant le code suivant, programmer la résolution par retour sur trace du problème du sudoku. L'entrée de la fonction `resoudre` est la liste de triplets présentées dans la question 1.

   On utilise une exception pour arrêter le programme dès qu'on a trouvé une solution.

```
exception Solution;;

let resoudre l =
  let g = init l in
  let rec aux i j =
    if i=9 then raise Solution
    else if g.(i).(j) <> Vide then if j=8 then aux (i+1) 0 (*Cas ou la case est pré-remplie*)
                                  else aux i (j+1)
    else begin (*Cas ou la case est vide*)
         for v = 1 to 9 do
            if possible g i j v then begin
                                       g.(i).(j) <- Pleine v;
                                       if j=8 then aux (i+1) 0
```

```
                                        else aux i (j+1)
                                        end
        done;
        g.(i).(j) <- Vide (*Vider pour éviter de confondre avec une case pré-remplie*)
        end
in
try aux 0 0; Printf.printf "Pas de solution" with Solution -> imprime g;;
```

2 Problème 2

1. On veut que soit X_1 et Z_1 soient vrais, soient X_1 et Z_1 soient faux. On veut donc satisfaire la formule suivante : $A = (X_1 \wedge Z_1) \vee (\neg X_1 \wedge \neg Z_1)$.
   Remarque : écrire $A \equiv \top$ est faux, A n'est pas une tautologie. Écrire $A = V$ n'est pas "homogène" au sens de la logique. La bonne manière de le dire est qu'on cherche v (la valuation) telle que $[|A|]_v = V$ ou alors qu'on veut satisfaire A.
   Si vous avez du mal à comprendre l'intuition derrière cette formule, dites vous qu'une valuation c'est un peu comme un univers (au sens de monde parallèle). Dans un univers les gens mentent tous, dans un univers les gens disent tous la vérité. Il existe aussi des univers où les gens disent ce qu'ils veulent. Dans cet exercice, on s'assure d'être dans un univers qui respecte les coutumes, donc qui rend A vrai. Tous les exercices de logique fonctionnent sur ce principe.
2. $X_1 = V \vee C$ et $Z_1 = \neg V$ (le choix de V comme variable n'est pas très malin vu le risque de confusion avec le vrai)
   Remarque : ici on définit ce que valent les propositions X_1 et Z_1. Précèdemment on écrivait les choses en fonction d'elles, mais maintenant on sait ce qu'elles valent
3. En remplaçant dans A, on obtient :

   $A = ((V \vee C) \wedge \neg V) \vee (\neg(V \vee C) \wedge \neg\neg V)$
   $\equiv ((V \wedge \neg V) \vee (C \wedge \neg V)) \vee (\neg V \wedge \neg C \wedge V)$ Distribution de "ou" sur "et" à gauche et lois de De Morgan à droite
   $\equiv (\bot \vee (C \wedge \neg V)) \vee \bot$ Antilogie classique
   $\equiv (\bot \vee (C \wedge \neg V))$ règle de calcul avec bottom
   $\equiv C \wedge \neg V$

   Arrivés à cette forme, on peut conclure : si on veut que A soit satisfaite, il faut que C soit affecté à vrai et V soit affecté à faux. Donc le village est dans les collines.
4. Même principe : $B = (X_2 \wedge Y_2 \wedge Z_2) \vee (\neg X_2 \wedge \neg Y_2 \wedge \neg Z_2)$
5. $X_2 = G \wedge D$, $Y_2 = M \rightarrow \neg D$, $Z_2 = G \wedge \neg M$.
6.

G	M	D	X_2	Y_2	Z_2	B
V	V	V	V	F	F	F
V	V	F	F	V	F	F
V	F	V	V	V	V	V
V	F	F	F	V	V	F
F	V	V	F	F	F	V
F	V	F	F	V	F	F
F	F	V	F	V	F	F
F	F	F	F	V	F	F

   On peut voir qu'il existe deux valuations satisfaisant B. Les deux valuations ont en commun de rendre D vraie. Donc on peut prendre le chemin de droite.
7. Parmi les deux valuations précédentes, une est un cas où tout le monde dit la vérité (la première) et l'autre est un cas où tout le monde ment (la deuxième).
   Si tout le monde ment, on peut aussi emprunter le chemin du milieu.