CS110 – Correction du devoir surveillé

Le barème final est précisé. Un point a été retiré à la question 7 et ajouté à la question 2. Les solutions données et commentées ici tentent d'être formulées le plus clairement possible, mais elles ne sont pas les seules et uniques solutions possibles.

1. (**2 pts**) La grille de sudoku est un tableau de 9 lignes, chaque ligne étant un tableau de 9 entiers (chaque entier représente l'état d'une case). Ou de manière équivalente, un tableau de 9 colonnes, chaque colonne étant un tableau de 9 entiers. Dans tous les cas, cela correspond à un tableau à deux dimensions de 9×9 cases. On n'oublie pas de préciser si le premier indice représente la ligne ou la colonne.

   ```
   type sudoku = tableau[1..9,1..9] d'entiers (* colonne, ligne *)
   ```

2. (**3 pts**) Il faut afficher une grille de sudoku. La procédure prendra un seul paramètre, qui est un paramètre d'entrée et sera donc passé par valeur. (1 pt pour cette explication).

   On doit pour chaque ligne, *afficher le contenu de cette ligne*, puis afficher le caractère `fdl` pour passer à la ligne. Il y a 9 lignes, numérotées de 1 à 9, on utilise donc une boucle `pour` avec un compteur de boucle `l` qui représente le numéro de la ligne que l'on doit afficher.

   ```
   pour l de 1 à 9 faire
        afficher la ligne l
        passer à la ligne
   fin pour
   ```

   Comment afficher une ligne `l` donnée ? Il faut afficher successivement la valeur des 9 cases de la ligne (ou un point si elle est nulle). Le traitement à répéter est *regarder la valeur de la case, afficher sa valeur ou un point si elle est nulle*. Ce traitement doit être répété pour chacune des cases de la ligne. On utilise à nouveau une boucle `pour`.

   ```
   pour c de 1 à 9 faire
          regarder la valeur de la case,
            afficher sa valeur ou un point si elle est nulle
   fin pour
   ```

   Au final, cela nous donne :

   ```
   procédure affiche_sudoku(s:sudoku)
   (* affiche le sudoku s *)
   var l,c,v:entier
   début
       pour l de 1 à 9 faire
            pour c de 1 à 9 faire
                 v ← s[c,l]
                 si v = 0 alors
                        écrire('.')
                 sinon
                        écrire(v)
                 fin si
            fin pour
            écrire(fdl)
       fin pour
   fin
   ```

3. (**3 pts**) En étudiant l'énoncé, on comprend que l'on doit écrire une fonction qui prend trois paramètres d'entrée : une grille de sudoku `s`, une valeur `v`, un numéro de ligne `l`. La fonction renvoie une valeur booléenne. On regarde chacune des cases de la ligne et on fait en sorte qu'un booléen soit mis à vrai si on en trouve une égale à `v` et reste faux dans le cas contraire. On utilisera pour cela une boucle `pour`.

```
fonction est_dans_ligne(s:sudoku; v:entier; l:entier):booléen
(* la valeur v est-elle présente dans la ligne l du sudoku s ? *)
var c:entier
    trouvé:booléen
début
    trouvé ← faux
    pour c de 1 à 9 faire
         trouvé ← trouvé ou s[c,l] = v
    fin pour
    retour(trouvé)
fin
```

Le principe est le même pour la recherche dans une colonne. On utilise ici une boucle `tant que`. Nous devons alors modifier nous même la valeur de la variable `l`. Le plus logique est de fixer initialement sa valeur à celle de la première ligne à considérer (donc ici 1) et à l'incrémenter *après* le traitement d'une case pour passer à la suivante. Nous terminerons quand nous aurons traité la dernière colonne, le dernier passage se fait avec `l` qui vaut 9.

```
l ← 1
tant que l≤9 faire
     tester l'élément de la colonne c, ligne l
     l ← l + 1
fin tant que
```

L'intérêt de la boucle `tant que` est qu'elle nous permet de stopper la recherche dès que (et si) on trouve la valeur `v` dans la colonne. Il y a donc une deuxième raison possible pour interrompre la boucle, le cas où on trouve la valeur cherchée. Il faut donc poursuivre la recherche tant que l'on est pas arrivé au bout de la colonne **et** qu'on n'a pas trouvé la valeur. Le traitement à répéter consiste simplement à voir si l'élément courant a la valeur cherchée.

```
fonction est_dans_colonne(s:sudoku; v:entier; c:entier):booléen
(* la valeur v est-elle présente dans la colonne c du sudoku s ? *)
var l:entier
    trouvé:booléen
début
    trouvé ← faux
    l ← 1
    tant que non trouvé et l≤9 faire
         trouvé ← s[c,l] = v
         l ← l + 1
    fin tant que
    retour(trouvé)
fin
```

4. (**4 pts**) Il faut bien comprendre l'énoncé : on nous donne une grille, une valeur `v`, une position dans la grille (valeurs `l` et `c`) et on veut déterminer si la valeur `v` se trouve déjà dans le bloc auquel appartient la position repérée par `l` et `c`. On suggère de commencer par déterminer les coordonnées du coin supérieur gauche du bloc en question, puis de tester toutes les cases de ce bloc (on rappelle que `div` représente la division entière). (1 pt pour ce premier sous-problème).

   On pourrait utiliser des boucles `tant que` pour stopper la recherche dès que l'on a trouvé la valeur si elle est présente, mais cela alourdirait la fonction et n'en vaut pas vraiment la peine. Deux boucles `pour` imbriquées permettent de parcourir les 3 lignes et les 3 colonnes du bloc.

```
fonction est_dans_bloc(s:sudoku; v:entier; c,l:entier):booléen
(* v est-il dans le même bloc que (c,l) ? *)
var cb,lb:entier (* coin sup gauche du bloc *)
    ci,li:entier (* compteurs des boucles *)
    trouvé:booléen
```

```
début
    (* coin supérieur gauche du bloc de (c,l) *)
    cb ← (c-1) div 3 × 3 + 1
    lb ← (l-1) div 3 × 3 + 1

    (* on cherche dans le bloc *)
    trouvé ← faux
    pour ci de cb à cb+2 faire
         pour li de lb à lb+2 faire
              trouvé ← trouvé ou s[ci,li] = v
         fin pour
    fin pour
    retour(trouvé)
fin
```

5. (**2 pts**) Il faut renvoyer la valeur vrai si *la case est vide* **et** *v n'est pas dans la ligne* **et** *v n'est pas dans la colonne* **et** *v n'est pas dans le bloc*. On utilise les fonctions que l'on a définies (ou pas, mais dans ce cas on suppose que cela a été fait) dans les questions précédentes.

```
fonction peut_jouer(s:sudoku; v:entier; c,l:entier):booléen
(* peut jouer v en position (c,l)  ? *)
début
retour (
         s[c,l] = 0 et
         non est_dans_ligne(s,v,l) et
         non est_dans_colonne(s,v,c) et
         non est_dans_bloc(s,v,c,l)
       )
fin
```

6. (**4 pts**) D'après l'énoncé, la procédure contient trois parties :
   (a) des traitements initiaux (demander la difficulté et initialiser la grille),
   (b) une boucle principale dans laquelle l'utilisateur peut placer des valeurs de manière répétée,
   (c) un traitement final consistant à afficher un message.

   Le corps de la boucle doit afficher le sudoku et lire trois valeurs `c`, `l` et `v`. Si `c` n'est pas égal à 0, c'est que l'utilisateur veut jouer un coup, on vérifie que le coup est possible (à l'aide de la fonction `peut_jouer` définie précédemment) et on le joue ou on affiche un message indiquant que le coup est impossible. Sinon (`c` égal à 0), on note que l'utilisateur veut abandonner.

   Quand la boucle doit-elle se terminer ? Quand le sudoku est complété ou que l'utilisateur abandonne. Si l'on utilise une boucle `tant que`, on doit continuer tant que le sudoku n'est pas complété **et** que l'utilisateur n'abandonne pas.

   Pour savoir si le sudoku est terminé, il suffit de garder à jour le nombre de cases libres, qui est connu au départ. On le décrémente à chaque coup joué. S'il devient nul, c'est que l'utilisateur a placé une valeur dans chaque case, et comme l'algorithme ne lui permet de le faire que si cela respecte les règles, c'est forcément que le sudoku est terminé.

   Le message (de congratulation ou de remontrances) sera affiché après la terminaison de la boucle. Le choix du message se fait en vérifiant laquelle des deux raisons possibles a provoqué la fin de la boucle.

```
procédure jeu
(* jeu de sudoku *)
var s:sudoku
    libres:entier   (* nombre de cases libres *)
    abandon:booléen
    c,l,v:entier
début
    écrire("Difficulté ")
    lire(libres)
```

```
    lit_sudoku(s, libres)

    abandon ← faux

    tant que libre>0 et non abandon faire
        affiche_sudoku(s)
        écrire("prochain coup : ")
        lire(c,l,v)
        abandon ← c=0
        si non abandon alors
            si peut_jouer(s,v,c,l) alors
                s[c,l] ← v
                libres ← libres - 1
            sinon
                écrire("coup impossible")
            fin si
        fin si
    fin tant que

    si abandon alors                (* ou libres ≠ 0 *)
         écrire("Et alors ?")
    sinon
         écrire("Bravo !")
    fin si
fin
```

7. (**2 pts**) Le coup à annuler est forcément le dernier coup joué. Cela correspond à une structure de pile (le premier sorti – coup annulé – est le dernier entré – coup joué). Que mettre dans la pile ? Simplement les positions jouées (couples (c,l)). À chaque coup joué, on empile la position, à chaque annulation on dépile et place un 0 dans la case correspondante.

   Les éléments de la pile sont par exmple du type `position` que l'on définit de la façon suivante :

```
type position = article
              c:entier
              l:entier
             fin article
```

   Il faudra déclarer la variable de type `pile` et commencer par l'initialiser à l'état de pile vide. En cas de demande d'annulation, vérifier que la pile n'est pas déjà vide avant de dépiler !

Quelques remarques générales

1. Choisissez des noms de variables "parlants", pas du genre `a` ou `b`. Cela est plus clair, vous voyez mieux ce que vous faites (et moi aussi) et vous risquez moins de vous embrouiller.
2. N'hésitez pas à placer des commentaires explicatifs dans vos programmes, en particulier quand ce que vous faites n'est pas absolument évident. Au besoin, précédez votre algorithme d'un paragraphe explicatif. Expliquez ce que l'on fait permet souvent de mieux le comprendre soi-même !
3. Répondez aux questions posées (!) Très peu d'entre vous ont justifié le mode de passage du paramètre de la question 2, qui tenait en une phrase et rapportait un point.
4. Ne réarrangez pas l'énoncé en fonction de ce qui vous arrange ou de ce que vous pensez "être mieux". Si vraiment vous avez des arguments pour procéder différement de ce qui est demandé, indiquez les.
5. Les erreurs sur les boucles sont souvent du type "on termine trop tôt ou trop tard". Une fois votre boucle écrite, exécutez la mentalement sur un exemple pour vérifier que tout se passe comme vous l'avez prévu.
6. Dernier conseil : il est important de s'entraîner à écrire des algorithmes. Il y a un monde entre comprendre un algorithme donné dans un corrigé et être capable de l'écrire soi-même. C'est en essayant, faisant des erreurs, puis comprenant ces erreurs que l'on assimile les choses. Si vous retravaillez sur vos exercices de TD, ne cédez pas trop vite à la tentation de regarder la solution. Écrivez-en une, essayez de vérifier qu'elle est correcte, corrigez les problèmes que vous rencontrez...