université de BORDEAUX

Stage Informatique

Réalisation de l'interface d'un sudoku en common Lisp

CLERET Anthony

Maître de stage
Irène Durand

Je tiens à remercier mon maître de stage Irène Durand pour m'avoir offert la possibilité de réaliser ce stage et pour son appui tout au long de ce dernier.

J'aimerai également remercier mes camarades pour aide.

Sommaire

Introduction

Actuellement en troisième année de licence informatique à l'Université de Bordeaux, je dois réaliser un stage en informatique dans le cadre de l'UE stage informatique permettant de valider la licence et d'approcher le monde du travail et/ou de la recherche. Pour le cinquième semestre d'informatique, l'Université de Bordeaux propose aux étudiants l'UE Programmation 3 dans laquelle les élèves approchent la programmation fonctionnelle et symbolique avec des techniques de programmation modulaire dans le langage informatique de très haut niveau Common Lisp. Mes motivations pour un stage en Common Lisp viennent principalement de mon intérêt pour ce langage ainsi que de mes brillants résultats dans la matière.

C'est dans l'optique de proposer aux étudiants de licence 3 informatique de cette rentrée 2015 un nouveau projet de programmation que Irène Durand, enseignante et chercheuse au LaBRI, m'a proposé un sujet de stage. Dans le cadre de l'UE Programmation Fonctionnelle et Symbolique dont elle est responsable, elle veut proposer aux élèves un projet à réaliser au cours du semestre. Cette année, elle soumettra à ses élèves la réalisation d'un jeu de sudoku, intéressant tant par son aspect programmation que par son aspect algorithmique.

Le sujet de mon stage est donc d'approcher le problème de la résolution d'une grille de sudoku et de réaliser une interface Web basée sur la bibliothèque hunchentoot pour permettre aux étudiants d'y faire tourner leur programme. Ainsi nous nous intéresserons dans un premier temps à l'affichage d'une grille en mode texte, puis nous développerons des algorithmes de résolution et nous finirons par l'interface Web.

I./ Le sudoku

Inspiré du carré magique, le sudoku est un jeu de grille à remplir inventé en 1979 par l'américain Howard Garns. Découvert en France au début des années 2000, le sudoku a vu sa popularité accroître au point de rejoindre les mots croisés dans la page des jeux de nos plus grands journaux.

a./ les règles

Communément, le format d'une grille est de 9 cases sur 9 avec 9 zones chacune au format 3 cases sur 3. Elle peut contenir au minimum une vingtaine de chiffres pour les grilles les plus difficiles, en général, moins il y a de chiffres sur la grille initiale, plus la grille est difficile. Une grille de sudoku doit respecter une règle simple:

"chaque ligne, chaque colonne et chaque zone doit contenir tous les chiffres de 1 à 9"

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

exemple de grille

On considère une grille valide si elle n'a qu'une seule et unique solution.

b./ les mathématiques du sudoku

Dès lors, le sudoku révèle des questions mathématiques auxquelles beaucoup de scientifiques se sont intéressées. Combien de grille existe-t-il? Doit-on considérer les grilles où les lignes/colonnes/zones/chiffres sont inversibles? Une grille peut-elle avoir plusieurs solutions? À partir de combien de chiffre peut-on résoudre une grille? Certaines de ces questions demeurent des

problèmes np-complets, sans preuve tangible où seul le résultat de recherches exhaustives approche les réponses.

c./ Les variantes du sudoku

Il existe plusieurs variantes du sudoku comme 4x4, 6x6, 16x16, sudoku killer, greater than, even/odd, 3doku, hypercube, magic sudoku, sudoku X, et bien d'autres. Autant de variantes intéressantes tant d'un point de vu mathématique qu'au niveau du jeu.

II./ Premier affichage en mode texte

a./ 9x9

Pour commencer le projet il nous faut dans un premier temps initialiser un tableau en variable globale. Plusieurs options s'offrent alors à nous : un tableau ou une liste de 3x3 contenant des sous-tableaux ou des sous-listes de 3x3, une liste de 9 sous listes contenant 9 éléments ou alors un tableau de 9x9. Nous opterons pour les tableaux plutôt que les listes car plus faciles d'accès quand ils ont plusieurs dimensions et car initialisables à partir de listes. Nous avons donc un tableau à deux dimensions de taille 9 par 9 initialisé à 0 à partir duquel nous réaliserons dans un premier temps un affichage simple grâce à une fonction d'affichage `print-grid`. Au début, nous définissions cette fonction en appelant uniquement la fonction `format` de la façon suivante:

```
(defun print-grid (grid)
  (dotimes (i 9)
    (if (zerop (mod i 3))
        (format t "  -------------------------~%"))
    (format t "  | ~d ~d ~d | ~d ~d ~d | ~d ~d ~d |~%"
            (aref grid i 0)
            (aref grid i 1)
            (aref grid i 2)
            (aref grid i 3)
            (aref grid i 4)
            (aref grid i 5)
            (aref grid i 6)
            (aref grid i 7)
            (aref grid i 8)))
  (format t "  -------------------------~%"))
```

Ainsi, les instructions suivantes retournent l'affichage de notre grille:

```
CL-USER> (defvar *grid*)
         (setf *grid* (make-array '(9 9) :initial-element 1))
         (print-grid *grid*)
  ------------------------
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  ------------------------
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  ------------------------
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  | 0 0 0 | 0 0 0 | 0 0 0 |
  ------------------------
NIL
```

Mais l'implémentation d'une telle fonction ne sert qu'à afficher des grilles de taille 9x9, il nous faudrait écrire une fonction différente pour chaque taille de grille. Nous allons donc utiliser de nouvelles variables globales contenant la taille de base m pour une grille nxn avec n tel que $n = m^2$, m dans N} :

- `*sqrt-size*` la taille de base m
- `*size*` la taille de la grille n (définie de la façon suivante `*sqrt-size*` x `*sqrt-size*`).

b./ généralisation de la taille

Nous afficherons notre grille à partir de ces variables afin de pouvoir construire de grilles de taille variable. Nous optimiserons également la présentation de la grille en ajoutant les coordonnées à la façon d'un échiquier, avec des lettres pour les colonnes et des nombres pour les lignes. Nous allons également ajouter une restriction sur la taille de la grille, elle ne pourra pas dépasser 25x25 premièrement car il n'y a que 26 lettres dans l'alphabet et deuxièmement parce qu'en général, la taille des jeux de sudoku n'excèdent pas 16x16. De plus, toujours pour des raisons de présentation, lorsqu'un nombre contenu dans une case dépasse 9, il sera remplacé par une lettre (comme en hexadécimal pour les 16x16 et de A à P pour les 25x25) et une case contenant un 0 (donc une case vide) sera affichée par un "_". Le code de la fonction sera donc le suivant:

```
(defun print-grid (grid)
  (when (not (check-is-valid-square grid));; vérifie que grid est un tableau carré
    (format t "ERREUR: taille non valide");; dont la taille est 2², 3², 4² ou 5²
    (return-from print-grid))
  (print-letters)                         ;; affiche le nom des colonnes
  (dotimes (i *size* (print-line))
    (when (zerop (mod i *sqrt-size*))
      (print-line))
    (print-number-line (1+ i))             ;; affiche le nom de la ligne
    (dotimes (j *size* (format t "|~%"))
      (if (zerop (mod j *sqrt-size*))
        (format t "| "))           ;; affiche un "|" toutes les *sqrt-size* cases
      (if (zerop (aref grid i j))
        (format t "_ ")            ;; affiche un "_" pour les cases vides
        (if (>= *size* 10)
            (if (>= (aref grid i j) 10) ;; affiche une lettre si le nombre >= 10
              (print-number-to-letter (aref grid i j))
              (format t "~d " (aref grid i j)))
            (format t "~d " (aref grid i j)))))))

;; affiche le nom des colonnes en fonction de la taille de la grille
(defun print-letters ()
  (format t "   ")
  (dotimes (i *size* (format t "~%"))
    (when (zerop (mod i *sqrt-size*))
      (format t "  "))
    (format t "~d " (code-char (+ i 97)))))

;; affiche une barre en fonction de la taille de la grille
(defun print-line ()
  (format t "   ")
  (dotimes (i (+ (* 2 *size*) (* 2 *sqrt-size*) 1) (format t "~%"))
    (format t "-")))

;; affiche le numéro de la ligne n
(defun print-number-line (n)
  (format t "~2d " n))

;; affiche la lettre majuscule correspondante à l'entier n à partir de 10
;; n=10 affiche A, n=11 affiche B, n=12 affiche C, etc.
(defun print-number-to-letter (n)
  (format t "~d " (code-char (+ n 55))))
```

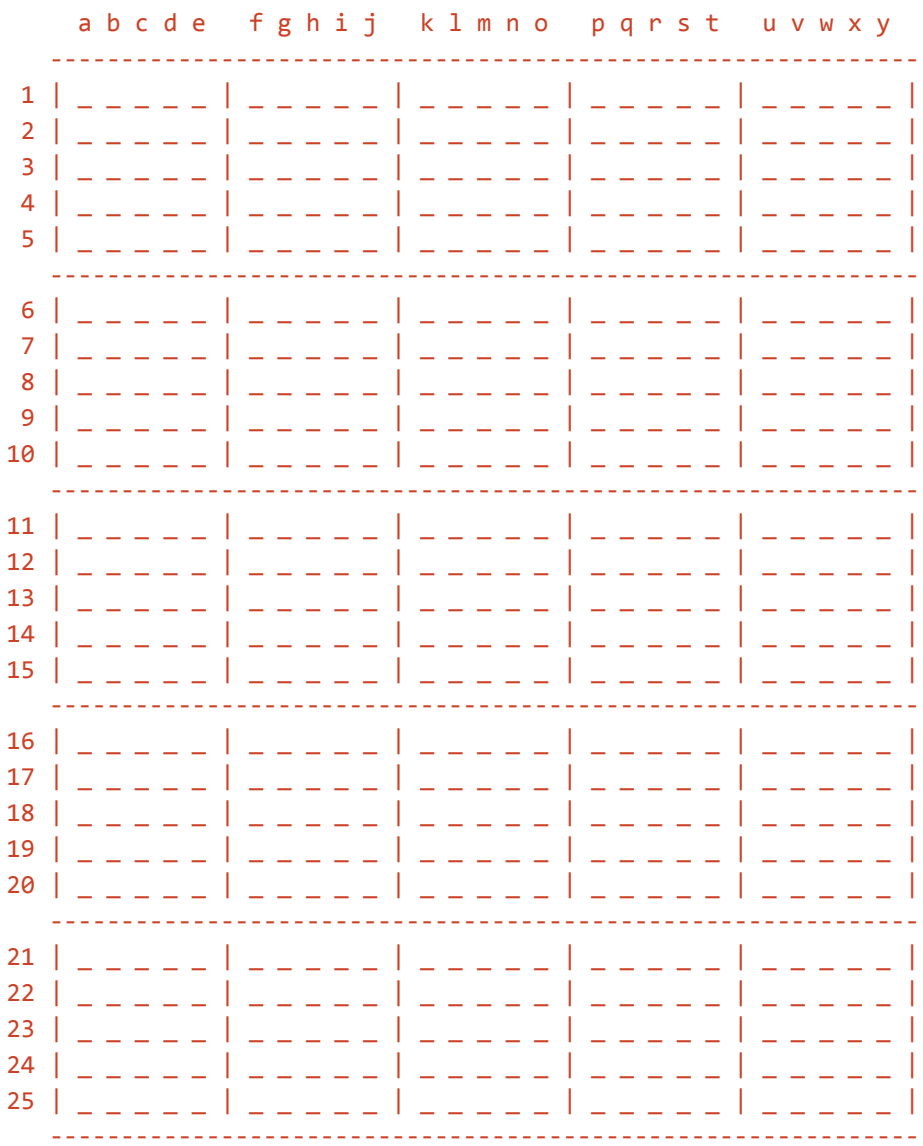
Ainsi, les instructions suivantes retournent l'affichage de notre grille:

```
CL-USER> (defvar *grid*)
         (setf *grid* (make-array '(4 4) :initial-element 0))
         (print-grid *grid*)

     a b   c d
   -------------
 1 | _ _ | _ _ |
 2 | _ _ | _ _ |
   -------------
 3 | _ _ | _ _ |
 4 | _ _ | _ _ |
   -------------
NIL
CL-USER> (defvar *grid*)
         (setf *grid* (make-array '(9 9) :initial-element 0))
         (print-grid *grid*)

     a b c   d e f   g h i
   -------------------------
 1 | _ _ _ | _ _ _ | _ _ _ |
 2 | _ _ _ | _ _ _ | _ _ _ |
 3 | _ _ _ | _ _ _ | _ _ _ |
   -------------------------
 4 | _ _ _ | _ _ _ | _ _ _ |
 5 | _ _ _ | _ _ _ | _ _ _ |
 6 | _ _ _ | _ _ _ | _ _ _ |
   -------------------------
 7 | _ _ _ | _ _ _ | _ _ _ |
 8 | _ _ _ | _ _ _ | _ _ _ |
 9 | _ _ _ | _ _ _ | _ _ _ |
   -------------------------
NIL
CL-USER> (defvar *grid*)
         (setf *grid* (make-array '(16 16) :initial-element 0))
         (print-grid *grid*)
```

```
    a b c d   e f g h   i j k l   m n o p
  -----------------------------------------
 1 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 2 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 3 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 4 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
  -----------------------------------------
 5 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 6 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 7 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
 8 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
  -----------------------------------------
 9 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
10 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
11 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
12 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
  -----------------------------------------
13 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
14 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
15 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
16 | _ _ _ _ | _ _ _ _ | _ _ _ _ | _ _ _ _ |
  -----------------------------------------
NIL
CL-USER> (defvar *grid*)
         (setf *grid* (make-array '(25 25) :initial-element 0))
         (print-grid *grid*)
```




```
NIL
```

Voyons maintenant comment afficher des grilles à partir de fichiers existants.

III./ Chargement et création d'une grille

Dans le but de créer un jeu de sudoku, nous avons besoin d'afficher des grilles à compléter. Pour cela nous allons dans un premier temps charger des grilles écrites dans une librairie de grille puis nous verrons par la suite comment créer nos propre grilles.

a./ chargement d'une grille

Nous aimerions charger des grilles contenues dans une librairie. Un tableau pouvant être initialisé à partir d'une grille de la forme `(make-array (list *size* *size*) :initial-contents l)` avec `l` une liste en 2 dimensions `*size*` par `*size*`, nous allons créer plusieurs fichiers contenant une grille sous forme de liste. Un fichier peut contenir par exemple:

```
((4 0 0 0)
 (0 3 0 0)
 (0 0 0 3)
 (0 0 4 1))
```

4x4-1.lisp

ou

```
((0 4 0 0 0 0 3 0 0)
 (0 0 6 8 2 0 0 4 7)
 (0 3 0 4 0 5 0 0 9)
 (0 8 4 0 0 7 9 0 0)
 (3 2 0 0 8 0 0 5 1)
 (0 0 1 3 0 0 4 6 0)
 (4 0 0 9 0 8 0 7 0)
 (7 1 0 0 5 6 8 0 0)
 (0 0 8 0 0 0 0 3 0))
```

9x9-1.lisp

Nous allons donc créer une fonction load-grid-from-file qui prend en paramètre une chaîne de caractère correspondant à l'endroit où se trouve le fichier et qui retourne un tableau contenant les éléments de sa liste. Par exemple:

```
CL-USER> (load-grid-from-file "../grid/9x9-1.lisp")
#2A((0 4 0 0 0 0 3 0 0)
    (0 0 6 8 2 0 0 4 7)
    (0 3 0 4 0 5 0 0 9)
    (0 8 4 0 0 7 9 0 0)
    (3 2 0 0 8 0 0 5 1)
    (0 0 1 3 0 0 4 6 0)
    (4 0 0 9 0 8 0 7 0)
    (7 1 0 0 5 6 8 0 0)
    (0 0 8 0 0 0 0 3 0))
```

Le code de la fonction est le suivant :

```
(defun load-grid-from-file (filename)
  (with-open-file (stream filename)
    (load-grid-from-stream stream)))

;; charge une grille à partir d'un stream
(defun load-grid-from-stream (stream)
  (let ((l (read stream)))
    (make-grid-from-list l)))

;; retourne une grille à partir d'une liste
(defun make-grid-from-list (l)
  (make-grid l))

;; retourne une grille à partir d'une liste ou crée une grille vide
(defun make-grid (&optional l)
  (if l
      (make-array (list *size* *size*) :initial-contents l)
      (make-array (list *size* *size*) :initial-element 0)))
```

En général, dans les jeux de sudoku que l'on trouve sur internet ou dans les applications mobiles, on ne demande pas au joueur de choisir sa grille mais plutôt sa difficulté. Nous avons décidé d'abandonner cette dernière contrainte pour avancer plus vite dans ce projet et seulement choisi de charger une grille aléatoirement parmi les grilles disponibles sans nous soucier de leur difficulté:

```
;; charge un fichier de la forme "../grid/sxs-n.lisp"
(defun load-grid-randomly()
  (let* ((r (random 8))      ;; on initialise un nombre aléatoire parmi le nombre
    (num (string (code-char (+ r 49))));; de fichier que l'on peut charger (ici 9)
      (size (string (code-char (+ *size* 48))));; 48 vaut "0" en ASCII
      (file ".lisp"))        ;; on initialise le nom du fichier
    (setf file (concatenate 'string "../grid/" size "x" size "-" num file))
    (load-grid-from-file file)))
```

Nous pourrons donc désormais charger des grilles aléatoirement dans notre jeu de sudoku. Néanmoins, le nombre de grille reste limité au nombre les fichier qu'il y a dans notre bibliothèque. Il nous faudrai un programme qui crée lui-même une grille aléatoire. C'est pourquoi nous allons développer dans un premier temps un programme qui génère des grilles pleines aléatoirement, puis nous verrons comment créer une grille à compléter.

b./ création de grille

Pour créer une grille de sudoku aléatoirement, l'idée est dans un premier temps d'écrire un programme qui génère une grille de sudoku de n'importe quelle taille et qui est complète et valide (chaque zone, ligne et colonne doit contenir tous les chiffres de 1 à `*size*`). Par la suite nous partirons de ces grilles complètes pour créer une grille de sudoku en supprimant plusieurs cases.

1./ grilles complètes

Pour générer une grille complète et valide nous partirons d'une grille vide que nous remplirons ligne par ligne. L'algorithme de la fonction `random-grid` est le suivant: on part d'une grille vide et on remplit ligne par ligne la grille d'une nouvelle ligne complète, valide et aléatoire grâce à la fonction `random-line`. L'algorithme de la fonction `random-line` est le suivant:

- on vide la ligne de la grille que l'on veut compléter
- pour chaque case de la ligne:
    - on récupère toutes les possibilités de la case
    - s'il n'y a aucune possibilité
        - on appelle récursivement random-line sur la ligne
    - sinon
        - on sélectionne aléatoirement une des possibilités
        - on l'affecte à la case

Pour éviter de boucler à l'infini lors de l'appel récursif, on utilisera un compteur qui servira à sortir des appels récursifs et à rappeler `random-grid` afin d'effacer la grille et de recommencer. Voici le code des fonctions `random-grid` et `random-line`:

```
;;retourne une grille de sudoku aléatoire et complète
(defun random-grid (grid)
  (empty-grid grid)            ;; on vide la grille grid
  (dotimes (i *size* grid)     ;; pour chaque ligne de grid
    (random-line grid i 0)))   ;; on appelle random-line
```

```
;; affecte à la ligne l de grid une ligne aléatoire si cpt <= *size* x *size*
(defun random-line (grid l cpt)
  (when (> cpt (* *size* *size*))
    (random-grid grid))
  (let ((r)
        (possibilities '()))
    (empty-line grid l)                          ;; on vide la ligne
    (dotimes (i *size* grid)              ;; on parcours chaque case de la ligne
      (setf possibilities (possible-numb grid l i))          ;; on récupère les
possibilités de la case
      (cond ((eq possibilities NIL)              ;; s'il n'y a pas de possibilités
             (random-line grid l (1+ cpt))       ;; on recommence sur la ligne l
             (return-from random-line)))  ;; en incrémentant cpt on sort
                                          ;; de la boucle si cpt non valide
      (setf r (random (length possibilities)));; sinon on récupère une possibilité
      (setf (aref grid l i) (nth r possibilities))))) ;; aléatoirement et on
                                                      ;; l'affecte
```

2./ grilles à compléter

Pour créer une grille de sudoku à compléter nous partirons donc d'une grille complète (générée grâce aux fonctions précédentes). Notre idée principale est la suivante: tant que la grille est résoluble, on efface un chiffre. Pour savoir si une grille est résoluble il nous faut donc dans un premier temps écrire un algorithme de résolution de grille.

a./ résolution d'une grille

Malgré les différentes tactiques pouvant être utilisées, nous en avons choisi deux principales à savoir "single in group" (seul dans une groupe) et "naked single" (choix unique). Les autres stratégies pouvant être implémentées plus tard.

Naked single

Cette tactique consiste à chercher les cases n'ayant comme valeur possible une seule et unique possibilité. Ici la case en jaune n'a comme possibilité que 3.

naked single

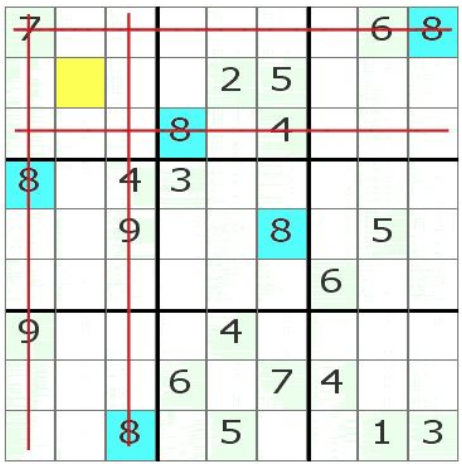
L'algorithme de la fonction tactics-naked-single qui prend donc une grille en paramètre est le suivant: tant qu'il y a une case avec une seule possibilité, on la lui affecte et on recommence. Voici le code:

```
;; retourne si une case de la grille n'a qu'une seule possibilité
(defun naked-single (grid i j)
  (and (zerop (aref grid i j))                    ;; la case est vide et il y a
       (= (length (possible-numb grid i j)) 1))) ;; exactement une possibilité

;; affecte les cases n'ayant qu'une seule possibilité
(defun tactics-naked-single (grid)
  (if (full-grid grid)           ;; si la grille est résolue
      grid                        ;; alors on la retourne
      (progn                     ;; sinon
        (dotimes (i (first (array-dimensions grid)) grid);; on parcours chaque
          (dotimes (j (second (array-dimensions grid)))  ;; case de la grille
            (when (naked-single grid i j) ;; s'il n'y a qu'une seule possibilité
            (setf (aref grid i j) (car (possible-numb grid i j))) ;; on affecte la
            (tactics-naked-single grid)))))))           ;; case et on recommence
```

Single in group

Dans le sudoku, on appelle un groupe une colonne, une ligne ou une zone. Cette technique élémentaire de résolution du puzzle consiste donc à chercher dans tous les groupes les possibilités qui n'y apparaissent qu'une seule et unique fois. Ici, dans la première zone, la case jaune est la seule case à avoir 8 comme possibilité.



single in group

L'algorithme de la fonction `tactics-single-in-group` qui prend donc une grille en paramètre est le suivant: tant qu'il y a une seule possibilité dans la case d'un groupe, on la lui affecte et on recommence. Pour vérifier s'il n'y a qu'une seule possibilité dans un groupe on appelle les fonctions `single-in-zone`, `single-in-col` et `single-in-line` qui retourne si la possibilité est unique dans la zone, colonne ou ligne. Voici le code:

```
;; retourne si nb n'est possible que dans la case l c de la zone de la case
(defun single-in-zone (grid nb l c)
  (let ((zone (zone l c)))                    ;; on récupère la zone de la case
    (dotimes (i (first (array-dimensions grid)) T) ;; on parcours la grille et on
                                                   ;; renvoie T
      (dotimes (j (second (array-dimensions grid)))
        (when (and (= zone (zone i j))         ;; si on est dans la zone la case
                   (not (coor-egal l c i j))   ;; n'est pas celle de l c et nb
                   (is-member nb (possible-numb grid i j))) ;; est une possibilité
          (return-from single-in-zone NIL))))))  ;; alors on sort et retourne NIL
```

```
;; retourne si nb n'est possible que dans la case l c de la colonne où se trouve la
case
(defun single-in-col (grid nb l c)
  (dotimes (i (first (array-dimensions grid)) T)   ;; on parcours la colonne et on
                                                    ;; renvoie T
    (when (and (not (equal (list l c) (list i c))) ;; si la case n'est pas l c
               (is-member nb (possible-numb grid i c)));; et nb est une possibilité
      (return-from single-in-col NIL))))     ;; alors on sort et on retourne NIL


;; retourne si nb n'est possible que dans la case l c de la ligne où se trouve la
case
(defun single-in-line (grid nb l c)
  (dotimes (i (first (array-dimensions grid)) T) ;; on parcours la ligne et on
                                                  ;; renvoie T
    (when (and (not (equal (list l c) (list l i)))    ;; si la case n'est pas l c
           (is-member nb (possible-numb grid l i)))  ;; et nb et une possibilité
      (return-from single-in-line NIL))))  ;; alors on sort et on retourne NIL

;; affecte les cases dont une possibilité n'y est possible que dans la zone, ligne
ou colonne
(defun tactics-single-in-group (grid)
  (if (full-grid grid) ;; si la grille est résolue
      grid             ;; alors on la retourne
      (progn           ;; sinon
       (dotimes (i (first (array-dimensions grid)) grid) ;; on parcours chaque
         (dotimes (j (second (array-dimensions grid)))   ;; case de la grille
           (let ((l (possible-numb grid i j))) ;; on récupère les possibilités
             (dotimes (nb (length l))          ;; et pour chaque possibilité
              (when (or
                     (single-in-zone grid (nth nb l) i j);; on vérifie si c'est la
                     (single-in-col grid (nth nb l) i j) ;; seule possibilité de la
                     (single-in-line grid (nth nb l) i j));; zone, ligne ou colonne
                (setf (aref grid i j) (nth nb l))         ;; on affecte la case
                (tactics-single-in-group grid)            ;; et on recommence
                (return-from tactics-single-in-group)))))))))
```

À partir de ce programme nous pouvons donc résoudre certaines grilles de sudoku. Nous allons maintenant pouvoir créer des grilles aléatoires à partir des ces algorithmes de résolution. L'idée est donc de partir d'une grille remplie puis de supprimer un case aléatoirement tant que la grille est résoluble. Il s'agit donc de définir le code retournant si une grille est résoluble ou non:

```
;; retourne si grid est résoluble
(defun is-solvable (grid)
  (let ((cp (copy-grid grid))) ;; initialise une copie de la grille
    (solve-grid cp)            ;; résout la copie
    (if (full-grid cp)         ;; retourne vrai si la grille est résolue
       T
       NIL)))

;; retourne la grille résolue
(defun solve-grid (grid)
  (solve-grid-aux grid (make-array (array-dimensions grid)))
  grid)

;; résout la grille initiale tant qu'elle est différente de la grille précédente
(defun solve-grid-aux (grid-init grid-prev)
  (if (cmp-grid grid-init grid-prev)
      grid-init   ;; retourne la grille si elle n'a pas changé
      (progn
       (setf grid-prev (copy-grid grid-init))  ;; sinon on la copie
       (tactics-naked-single grid-init)        ;; on résout notre grille avec
       (tactics-single-in-group grid-init)     ;; nos deux tactiques
       (solve-grid-aux grid-init grid-prev)))) ;; et recommence si la grille
                                                ;; a été modifiée
```

b./ fonction finale et efficacité

Maintenant que nous pouvons savoir si une grille est résoluble ou non grâce à nos tactiques, nous allons maintenant enfin pouvoir créer des grilles aléatoirement. L'idée est la suivante: on initialise une grille aléatoire pleine, et on supprime une case non vide aléatoire tant que la grille est résoluble. Voici donc le code de la fonction create-grid:

```
;; retourne une grille aléatoire
(defun create-grid ()
  (let ((grid (create-grid-aux (random-grid (make-grid)) 0))) ;; on part d'une grille
                                                              ;; aléatoire complète
    (format t "cases initiales : ~d~%" (square-numb grid)) ;; on affiche le nombre de case
    grid)) ;; et on retourne la grille

;; complète une grille aléatoirement tant que i différent de *size*
(defun create-grid-aux (grid i)
  (if (= i *size*) ;; si i atteint *size*
      grid         ;; on retourne la grille
      (progn                                  ;; sinon
        (let* ((rcase (random-full-case grid)) ;; on choisit une case aléatoirement
          (val (aref grid (first rcase) (second rcase))))    ;; on récupère sa valeur
          (setf (aref grid (first rcase) (second rcase)) 0)  ;; on supprime la case
          (if (is-solvable grid)                             ;; si la grille est résoluble
            (create-grid-aux grid i)                         ;; alors on continue
            (progn (setf (aref grid (first rcase) (second rcase)) val) ;; sinon on annule
                   (create-grid-aux grid (1+ i))))))))));; et on recommence en incrémentant i
```

Initialement, notre programme pouvait créer 100 grilles en seulement 5 secondes mais avec une moyenne de 44 chiffres initiaux, moyenne trop élevée par rapport aux grilles de sudoku les plus faciles (environ 34). Désormais notre programme peut créer des grilles de difficulté intermédiaire voire difficile avec une moyenne de 29 chiffres initiaux mais en plus de temps. Maintenant il faut environ 2 minutes pour créer 100 grilles.

IV./ Le jeu

Notre jeu de sudoku doit afficher une grille et proposer au joueur une grille qu'il doit remplir case par case pour gagner. Il doit donc avoir la possibilité de rentre les coordonnées d'une case ainsi que le nombre qu'il veut lui affecter. Le jeu pourra éventuellement retourner des erreurs en fonction du choix du joueur comme par exemple si le chiffre choisi est déjà présent dans la colonne de la case.

À ce stade nous avons donc écrit les fonctions d'affichage ainsi que de génération de grilles aléatoires. Nous ignorerons le choix de difficulté pour plus de simplicité. Notre jeu affiche dans un premier temps un message d'accueil puis crée une grille de sudoku grâce aux fonctions précédentes puis l'affiche et demande au joueur de faire un choix.

```
CL-USER> (sudoku)

 ---------------------------------------------------------------
                          SUDOKU 9x9
 ---------------------------------------------------------------
choisissez à tout moment : - n pour nouvelle partie
                           - r pour recommencer la partie
                           - q pour quitter
cases initiales : 32
     a b c   d e f   g h i
   -------------------------
 1 | _ _ 4 | _ 7 5 | _ _ _ |
 2 | _ _ _ | _ _ _ | 9 5 _ |
 3 | 9 _ 8 | _ _ 2 | 4 _ 7 |
   -------------------------
 4 | 7 _ _ | _ 3 _ | _ _ _ |
 5 | 5 _ _ | _ 9 _ | 7 _ 4 |
 6 | _ _ 2 | 7 4 1 | 3 _ _ |
   -------------------------
 7 | _ _ 6 | 4 _ 9 | _ _ 2 |
 8 | _ 3 5 | 1 _ 7 | _ _ _ |
 9 | _ _ _ | _ 6 _ | 5 7 _ |
   -------------------------
choisissez un nombre, une colonne et une ligne (ex 6 a 1): ▌
```

Lors du développement du jeu nous avons décidé de guider le joueur en étudiant chacun de ses choix, comme dans la plupart des jeux de sudoku en ligne. Ainsi, si le joueur choisit par exemple `5 h 3` dans la grille ci-dessus alors le choix est ignoré et le jeu affiche le message d'erreurs suivant:

```
choisissez un nombre, une colonne et une ligne (ex 6 a 1): 5 h 3
ERREUR: nombre 5 déjà dans la colonne H
ERREUR: nombre 5 déjà dans la ligne 3
ERREUR: nombre 5 déjà dans la zone 2
ERREUR: l'affectation de 5 cette case H 3 rend la grille non résoluble
```

Le jeu gère également les erreurs de type syntaxique comme le choix de lettres à la place de chiffres et inversement ainsi que les choix valides mais des cases qui sont protégées. Une case d'une grille est de type protégée si elle est non vide dans la grille initiale. La grille protégée est en fait une copie de la grille initiale et pour savoir si une case est protégée ou non on accède à la même case de la grille protégée et on regarde si elle est vide (vaut 0) ou non.

De plus le jeu offre la possibilité au joueur de choisir à tout moment de la partie de jouer une nouvelle partie en choisissant n, de recommencer la grille avec r, d'afficher la grille avec p ou encore de quitter la partie et revenir dans le prompt avec q. Voici donc le code de notre jeu de sudoku final:

```
(defgeneric grid (game))
(defgeneric protected-grid (game))

(defclass game () ;; on crée un nouveau type game
  ((grid :accessor grid :initarg :grid)         ;; il possède une grille de sudoku
   (protected-grid :reader protected-grid :initarg :protected-grid)))
                 ;; et une grille contenant ses cases protégées

;; pour créer un jeu on copie la grille initiale dans la grille protégée
(defun make-game (grid)
  (make-instance 'game :protected-grid grid))

;; initialise le jeu
(defgeneric init-game (game))
(defmethod init-game ((game game))
  (setf (grid game) (copy-grid (protected-grid game))))

;; initialise et lance le jeu
(defgeneric run-game (game))
(defmethod run-game ((game game))
  (init-game game)
  (play game))

;; la fonction principale que l'on lance dans le prompt pour commencer la partie
(defun sudoku ()
```

```
  (message-begin) ;; affiche un message de départ
  (let ((grid (create-grid))) ;; crée un grille aléatoire
    (run-game (make-game grid)))) ;; lance la partie avec cette grille

;; propose au joueur un choix, gère les erreurs en fonction du choix ou affecte le
choix à la grille s'il est correct et recommence
(defun play (game)
  (let ((grid (grid game))) ;; on récupère la grille du jeu
    (print-grid grid)       ;; on l'affiche
    (format t "choisissez un nombre, une colonne et une ligne (ex 6 a 1): ")
    (let ((nb (read)))      ;; si le premier choix est n, r ou q
      (cond ((eq nb 'n)     ;; si n on lance une nouvelle grille
             (run-game (make-game (create-grid))))
            ((eq nb 'r)     ;; si r on recommence la grille courante
             (run-game game))
            ((eq nb 'q)     ;; si q on retourne dans le prompt
             (format t "vous avez quitté la partie")
             (return-from play))
            ((eq nb 'p)     ;; si p on affiche la grille
             (play game)))
      (let* ((col (read))   ;; sinon on récupère les 2e et 3e choix
             (line (read))
             (err (manage-errors game nb col line))) ;; affiche les erreurs
       (if err              ;; s'il y a une erreur
           (play game)      ;; on ignore et recommence play
           (progn           ;; s'il n'y a pas d'erreur
             (play-in-square grid (1- line) (letter-int col) nb) ;; on affecte
             (if (full-grid grid) ;; et si la grille est pleine
                (end-game game)   ;; alors termine la partie
                (play game))))))))

;; termine la partie
(defgeneric end-game (game))
(defmethod end-game ((game game))
  (format t "
 ------------------------------------------------------
|  Félicitation, vous avez terminé la grille de sudoku  |
 ------------------------------------------------------
Choisissez q pour quitter
Choisissez n pour une nouvelle partie
mon choix : ")
  (let ((choice (read)))
    (cond ((eq choice 'n)
           (run-game (make-game (create-grid))))
          ((eq choice 'r)
           (run-game game))
          ((eq choice 'q)
           (error "vous avez quitté la partie"))))
  (end-game game))
```

gestion des erreurs

```
;; affiche les erreurs en fonction du choix du joueurs
(defun manage-errors (game nb col line)
  (let ((err nil))
    (if (or (not (member col (firsts-elem *list-alpha* *size*)))
            (not (member nb *squares*))
            (not (member line *squares*)))
        (progn (format t "ERREUR: nombre ~d ou case ~d ~d non valide~%" nb col line)
               (setq err t))
        (progn (when (in-col (grid game) (position col *list-alpha*) nb)
                (format t "ERREUR: nombre ~d déjà dans la colonne ~d~%" nb col)
                (setq err t))
               (when (in-line (grid game) (1- line) nb)
                (format t "ERREUR: nombre ~d déjà dans la ligne ~d~%" nb line)
                (setq err t))
               (when (in-zone (grid game) (1- line) (position col *list-alpha*) nb)
                (format t "ERREUR: nombre ~d déjà dans la zone ~d~%" nb
                       (zone (1- line) (position col *list-alpha*)))
                (setq err t))
                 (when (not (zerop (aref (protected-grid game) (1- line) (letter-int
col))))
                (format t "ERREUR: case ~d ~d protégée~%" col line)
                (setq err t))
               (let ((cp (copy-grid (grid game))))
                      (setf (aref cp (1- line) (letter-int col)) nb)
                      (when (not (is-solvable cp))
                        (format t "ERREUR: l'affectation de ~d cette case ~d ~d rend
la grille non résoluble~%"
                                nb col line)
                        (setf err T)))))
    err))
```

Notre programme contient beaucoup d'autres fonctions dont nous ne parlerons pas car il s'agit de bouts de code simple qui ne servent qu'à rendre plus lisible ou encore à éviter la duplication de code. La définition de ces fonctions de trouve dans le code du projet en annexe.

IV./ Bilan

S'agissant d'une part du bilan professionnel, je tire de cette expérience un constat bien plus que positif. J'ai pu en effet appréhender et avoir une nouvelle approche d'un langage informatique que j'ai déjà rencontré au cours de mon cursus universitaire. Ainsi, ce nouveau projet a été des plus intéressant dans le sens où il m'a permis de faire évoluer la conception de certains éléments notamment l'usage de nouvelles fonctions relatives au Common Lisp, le typage ou encore en programmation fonctionnelle. De plus, la réalisation de ce stage s'est révélé être pour moi une véritable opportunité puisqu'il m'a offert la possibilité d'utiliser mes connaissances théoriques dans ce langage et de les mettre en oeuvre. Autrement dit, j'ai pu allier à la fois le côté théorique au côté pratique. D'ailleurs, l'aspect mathématique et algorithmique reste ce qui m'a le plus attiré dans l'accomplissement de cet exercice. En effet, étant moi-même amateur de Sudoku, j'ai pu interchanger ma place habituelle de joueur à celle de concepteur ce qui reste, sous certains aspects, parfois quelque peu complexe.

S'agissant du bilan personnel, encore une fois celui-ci reste positif. En effet, même s'il est vrai que j'ai pu avoir quelques doutes quant à mes capacités à gérer les différentes situations et notamment à mener seul un projet d'une telle envergure, cette expérience m'a permis de prendre des initiatives et de mieux m'organiser. Effectivement, le travail en autonomie et la recherche ont été pour moi les principaux mots d'ordre de ce stage, tout en bénéficiant de l'appui de mon professeur. Cependant, j'ai du me confronter à un certains nombre de difficultés tout au long de ce projet. En outre, bien qu'ayant travaillé sur la réalisation d'une page HTML, permettant l'affichage d'un jeu de Sudoku, ainsi que sur l'écriture du présent rapport en LaTeX, ces éléments n'ont pu aboutir car j'ai du faire face à de multiples difficultés. Des difficultés, liées à mon absence d'expérience dans le Web mais aussi techniques puisque certains outils informatiques notamment le logiciel texmaker dont le fonctionnement s'est avéré complexe ne dispose pas des packages nécessaires et des droits suffisants à leur installation. Néanmoins, je reste plus que motivé quant à l'approfondissement des ces éléments.

Autant d'aspects qui m'encouragent à poursuivre dans le domaine du développement informatique et de confirmer mon projet tant sur le plan professionnel que personnel.

Bibliographie

http://fr.wikipedia.org/wiki/Sudoku
http://www.e-sudoku.fr/
http://fr.wikipedia.org/wiki/Math%C3%A9matiques_du_Sudoku
http://www.u-bordeaux.fr/
http://dept-info.labri.fr/~strandh/Teaching/MTP/Common/Book/HTML/programmation.html
http://www.funmeninges.com/sudoku-tactiques_cs-3-1.html