# TP 2— SUDOKU

http://www.liafa.jussieu.fr/~nath/tp2/tp2.pdf

Les questions sont les bienvenues et peuvent être envoyées à nathanael.fijalkow@gmail.com.

L'objectif de ce TP est de résoudre un sudoku de manière automatique, et disons-le tout de suite, peu subtile, par la méthode du backtracking (d'aucuns diraient "retour sur trace"). Pour cela, une fois n'est pas coutume, nous utiliserons un peu de programmation impérative, par opposition à récursive.

La première section est une mise au point sur un écueil courant : les fonctions à plusieurs paramètres et le placement des parenthèses. La deuxième section est une mise au point sur la programmation impérative.

## 1 TRAITEMENT DES FONCTIONS À PLUSIEURS PARAMÈTRES

Une fonction est une valeur comme les autres. Elle peut par exemple être passée en argument d'une autre fonction, ou renvoyée comme valeur de retour. Ici, `deux_fois` prend une fonction `f` en paramètre et renvoie une autre fonction (`fun x -> ..`).

```
# let deux_fois f = (fun x -> f x + f x);;
# deux_fois (fun n -> n + 1) 3 ;;
- :  int = 8
```

Si l'on fait un peu attention, on peut aussi voir `deux_fois` comme une fonction qui prend deux arguments, $f$ et $x$, et renvoie $f(x) + f(x)$. On peut d'ailleurs écrire (et c'est strictement équivalent) :
`let deux_fois f x = f x + f x`.

Cette méthode permet d'exprimer des fonctions à plusieurs arguments. On peut la généraliser à un nombre quelconque (mais fixé à l'avance) d'arguments. Elle s'appelle la *curryfication*, et correspond à l'isomorphisme, en mathématiques, entre $(A \times B) \to C$ et $A \to (B \to C)$.

**Remarque :** Avec les fonctions à plusieurs arguments, on voit apparaître des types de la forme `int -> int -> int`. Dans ces cas là il faut lire (la flèche est associative à droite) `int -> (int -> int)`.

**Notations** On peut écrire `let f x = a` à la place de `let f = fun x -> a`. Selon le même principe, on peut simplifier l'écriture de fonctions à plusieurs paramètres :

- on peut compresser une suite de `fun` : le code `fun x -> fun y -> z` s'écrit aussi `fun x y -> z`.
- on peut faire rentrer plusieurs `fun` dans un `let` : `let f = fun x y -> z` s'écrit aussi `let f x y = z`.

Il faut bien comprendre que ces notations représentent exactement le même code.

### 1.1 APPLICATION PARTIELLE

Il n'y a pas à proprement parler de "fonction à deux arguments" en Caml. Il y a par contre des fonctions qui prennent un argument, puis renvoient une fonction qui prend un deuxième argument et renvoie un résultat. En particulier, fournir un seul argument à une fonction Caml est toujours défini, vous n'aurez jamais d'erreur du style "Vous n'avez pas fourni assez d'arguments".

```
# let somme x y = x + y;;
# let somme' x = fun y -> x + y;;
# let somme_3 = somme' 3;;
# somme_3 2;;
- int :  5
```

Par exemple, dans le code ci-contre, les fonctions `somme` et `somme'` sont exactement identiques, mais la deuxième permet mieux de comprendre ce que signifie `somme 3` : c'est la fonction qui, quand on lui donne un paramètre, lui ajoute 3. On parle dans ce cadre d'*application partielle* : on a donné seulement une partie des arguments à la fonction.

### 1.2 PLACEMENT DES PARENTHÈSES

La source d'erreur est le fait que le placement des parenthèses en Caml est *différent* de la notation mathématique habituelle.

Les parenthèses utilisées autour des expressions n'ont *pas de signification*. Elles servent uniquement à éliminer les ambiguités, comme en mathématiques (différence entre $1 + 2 * 3$ et $(1 + 2) * 3$).

Pour appeler une fonction `f` avec un argument `x`, on écrit simplement `f x`. Vous pouvez aussi écrire `(f x)`, `(f)x`, `f(x)` ou `(f)((x))` si cela vous fait plaisir, mais contrairement à la notation mathématique, les parenthèses autour du `x` ne sont *pas* nécessaires, et je vous déconseille de les mettre.

Le problème de la notation mathématique est son extension aux appels de fonction à plusieurs arguments, comme par exemple `g x y` : on donne à la fonction `g` l'argument `x`, puis `y`. Si l'on veut être précis, on dit que l'application de `x` à `g` renvoie une fonction (`g` est de la forme `fun a -> fun b -> c`), et que c'est à cette fonction qu'on applique `y`. On pourrait donc aussi écrire `let h = g x in h y` ou `(g x) y`.

C'est ici que la notation mathématique à laquelle vous êtes habitués risque de vous faire faire des erreurs. `g x y` est équivalent à `(g x) y`, mais **pas** à `g(x y)` ! Cette deuxième écriture signifie qu'on applique `y` à `x`, et qu'on donne le résultat à `g`. Par contre, si vous voulez écrire $f(g(x))$, c'est bien `f (g x)` et pas `f g x` !

▷ **Question 1.** Que déduire du code ci-contre? ◁

```
let f x y = () ;;
let a = ref 0 in f (a := 1) (a := 2) ; !a ;;
```

# 2 Rappels de programmation impérative

## 2.1 Champs modifiables

En `caml`, les variables ne sont pas *modifiables* : une fois qu'une variable a été déclarée par un `let`, sa valeur ne change pas (jusqu'à la déclaration suivante).

Si `'a` est un type `caml`, `'a ref` désigne le type des *références* contenant une valeur de type `'a`. On peut voir une référence `x` comme un tiroir : on peut l'ouvrir pour regarder à l'intérieur (avec `!x`, on voit un objet de type `'a`), ou bien changer son contenu pour y mettre la valeur `v` de type `'a`, (avec `x := v`).

Pour créer une nouvelle référence, on utilise la fonction `ref` en lui donnant une valeur de départ (la valeur que contiendra le tiroir avant que son contenu ne soit modifié). Attention à la subtilité : quand on utilise l'opérateur `:=`, on change le *contenu* de la référence, pas la référence elle-même (qui désigne toujours le même tiroir) !

```
let test =
  let a, b = ref 1, ref 2 in
  let c, d = a, ref a in
  a := !a + !b; d := c; !(!d) + !a;;
```

▷ **Question 2.** Quelle est la valeur de la variable `test` ? ◁

Les cases des tableaux et des chaînes de caractères sont aussi des champs modifiables : `tab.(i)` et `str.[i]` permettent d'obtenir la valeur en i-ème position (en partant de 0) du tableau `tab` et de la chaîne `str`. Pour les modifier on n'utilise pas "`:=`" mais "`<-`" : par exemple `tab.(i) <- v`.

## 2.2 Exceptions

Les exceptions sont une manière d'interrompre une partie d'un programme en cas d'erreur. Par exemple, si vous avez une formule mathématique à calculer, et qu'en plein calcul vous vous apercevez que vous devez diviser 0, vous allez vous arrêter et vous plaindre que la formule n'est pas bien définie. `caml` sait faire pareil.

Les expressions sont des objets de type `exn` qui ressemblent beaucoup aux types sommes définis dans le précédent TP : ce sont des constructeurs, qui peuvent comporter des arguments, et sont déclarés par le mot-clé `exception`.

Quand on a trouvé une erreur, on peut *lancer* une expression avec la fonction `raise` : elle prend une expression en paramètre, et interromp le calcul (en particulier, tout ce qui devait se passer ensuite dans le programme n'est pas exécuté).

```
#exception Erreur of string;;
let boum () = raise (Erreur "boum");;

try (boum (); "message") with
 | Exit -> "sortie"
 | Erreur message -> "erreur :  " ^ message
 | _ -> "exception inconnue"
```

Cela permet de faire des erreurs qui stoppent complètement le calcul. Parfois, on voudrait plutôt détecter l'erreur et utiliser une solution adaptée pour continuer le programme (par exemple si l'erreur est "plus de papier dans l'imprimante", il suffit de demander à l'utilisateur de rajouter du papier avant de continuer, au lieu d'annuler complètement l'impression en cours). On peut *rattrapper* une exception avec la construction `try <expr> with <filtrage>`. Cela se présente un peu comme un `match ... with`, mais le comportement est différent :

- si l'évaluation de `<expr>` ne provoque aucune exception, on renvoie sa valeur
- sinon, on effectue le filtrage sur la valeur de l'exception envoyée

▷ **Question 3.** Écrire une fonction `existe` avec une boucle `for`, tel que `existe elt tableau` teste l'existence de `elt` dans le tableau `tableau` en parcourant le moins possible d'éléments du tableau. ◁

▷ **Question 4.** Quel est le type de `raise` ? Pourquoi ? ◁

### 2.3 Dangers

On veut créer un tableau à deux dimensions, sans utiliser la fonction déjà toute faite `Array.make_matrix`.

▷ **Question 5.** Quel est le problème avec `mat` ?
Coder (correctement) `nouvelle_matrice` avec une boucle. ◁

```
let nouvelle_matrice n p x =
  Array.make n (Array.make p x);;
let mat = nouvelle_matrice 3 3 0;;
mat.(0).(1) <- 2; mat;;
```

## 3 Sudoku

Les sudokus classiques sont des grilles $9 \times 9$ où chaque case est soit blanche, soit contient un chiffre parmi $\{1, \ldots, 9\}$. Par convention, les cases blanches contiennent des $0$. L'objectif est de remplir les cases blanches, de manière à ce que sur chaque ligne, sur chaque colonne, et sur chacun des $9$ carrés $3 \times 3$, chaque chiffre apparaît une fois et une seule. Ce sont des sudokus d'ordre $3$.

Un sudoku d'ordre $n$ est une grille $n^2 \times n^2$ où chaque case est soit blanche, soit contient un nombre parmi $\{1, \ldots, n^2\}$. L'objectif est de remplir les cases blanches, de manière à ce que sur chaque ligne, sur chaque colonne, et sur chacun des $n^2$ carrés $n \times n$, chaque chiffre apparaît une fois et une seule.

Pour résoudre un sudoku, nous allons utiliser la méthode du backtracking. L'entrée est un entier $n$ et une grille $n^2 \times n^2$. **Cette grille ne sera pas modifiée**. On procède récursivement, en maintenant une liste des choix effectués, sous la forme de triplet $(i, j, k)$ : ligne $i$, colonne $j$, on place l'entier $k$. L'algorithme procède ainsi : si les conditions du sudoku ne sont pas violées, alors faire un nouveau choix, sinon revenir en arrière, c'est-à-dire modifier le dernier choix qui peut l'être et supprimer les choix faits après celui-ci. S'il n'est pas possible de revenir en arrière, c'est qu'il n'y a pas de solution.

```
let resoudre_sudoku n grille =
let nc = n * n in
let rec boucle liste_choix =
try
 if correct n nc grille liste_choix
  then boucle (choix_suivant nc grille liste_choix)
  else boucle (changer_choix nc grille liste_choix)
with
  | Fini -> retourner_solution nc grille liste_choix
  | Echec -> failwith "pas de solution"
in boucle [] ;;
```

▷ **Question 6.** Coder la fonction `choix_suivant`. Elle déclenche l'exception `Fini` s'il n'y a plus de case libre. ◁

▷ **Question 7.** Coder la fonction `changer_choix`. Elle déclenche l'exception `Echec` s'il n'est pas possible de revenir en arrière. ◁

▷ **Question 8.** Coder la fonction `correct`. Remarquer qu'il est inutile de tester toute la grille, seulement une ligne, une colonne et un carré. Coder la fonction `retourner_solution`. Tester votre code, par exemple sur une grille vide. ◁

▷ **Question 9.** Modifier à peine le code pour rendre toutes les solutions (au lieu de rendre la première trouvée). ◁

▷ **Question 10.** Fixons une grille d'ordre $n$ possédant $k$ cases blanches. Une configuration est une liste de triplets, remplissant certaines des cases blanches de la grille. Elle est valide si elle ne viole pas les conditions du

sudoku une fois insérée dans la grille. L'ensemble des configurations valides peut être représenté de manière à interpréter l'algorithme implémenté : plaçons tout en haut la configuration vide (liste vide). Étant donné une configuration valide $c$, on lui associe des fils, qui sont les configurations valides qui étendent la configuration $c$ d'exactement un triplet. Ceci définit (il faudrait être plus précis) l'arbre des configurations. Que représentent les feuilles ? Quelle est la taille de cet arbre, combien y a t-il de feuilles (en fonction de $n$ et $k$) ? Dernière question : l'algorithme implémenté parcourt les configurations. Décrire le parcours de l'arbre qu'il effectue. Votre modification du code (question précédente) visite t-il toutes les configurations ? ◁

▷ **Question 11.** La fonction `resoudre_sudoku` est-elle récursive terminale? ◁

▷ **Question 12.** De nombreuses optimisations sont possibles, dont certaines s'appuie sur une étude fine de l'arbre des configurations. Proposer et implémenter quelques optimisations. ◁

## 4 Question difficile

Cette dernière partie est une petite incursion vers ce que cache la programmation fonctionnelle (à savoir le lambda-calcul). L'entier de Church noté $\hat{n}$ est la fonction qui à $f$ associe $f^n$ (l'itérée $n$ fois de $f$).

On peut le définir en `caml` par :
Ainsi `church n` retourne $\hat{n}$.

```
let rec church n f x =
 if n = 0 then x
 else church (n-1) f (f x);;
```

▷ **Question 13.** Quel est le type de $\hat{n}$? ◁

▷ **Question 14.** Coder $\hat{0}, \hat{1}$ et $\hat{2}$. ◁

▷ **Question 15.** Coder la fonction `eval` qui à $\hat{n}$ associe $n$. ◁

▷ **Question 16.** Coder la fonction `succ` qui à $\hat{n}$ associe $n \hat{+} 1$. ◁

▷ **Question 17.** Coder les fonctions `add`, `mult` et `exp` qui calculent la somme, le produit et l'exponentiation de deux entiers de Church. ◁