

Correction du DS 3

Julien REICHERT

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici. De même, les exercices issus des TD et TP ont leur correction déjà publiée.

Exercices

Exercice 1 :

```
let rec ajout (cle, valeur) liste = match liste with
| [] -> [(cle, valeur)]
| (c, v)::q when c = cle -> (cle, v +. valeur)::q
| (c, v)::q -> (c, v) :: ajout (cle, valeur) q

let rec toutes_sommes liste = match liste with
| [] -> []
| (cle, valeur)::q -> ajout (cle, valeur) (toutes_sommes q)

let toutes_sommes_dico d =
  let dico = Hashtbl.create 8 in
  let f cle valeur = match Hashtbl.find_opt dico cle with
  | Some total -> Hashtbl.replace dico cle (valeur +. total)
  | None -> Hashtbl.add dico cle valeur
  in Hashtbl.iter f d; dico

struct entree_dico { int cle ; double valeur ; };
typedef struct entree_dico entree;

entree* toutes_sommes(entree* dico, int taille, int* nouvelle_taille)
{
  entree* temporaire = malloc(taille * sizeof(entree));
  *nouvelle_taille = 0;
  for (int i = 0 ; i < taille ; i += 1)
  {
    int j = 0;
    while (j < *nouvelle_taille && dico[i].cle != temporaire[j].cle) j += 1;
    if (j == *nouvelle_taille)
    {
      temporaire[j].cle = dico[i].cle;
      temporaire[j].valeur = dico[i].valeur;
      *nouvelle_taille += 1;
    }
    else
    {
      temporaire[j].valeur += dico[i].valeur;
    }
  }
  entree* reponse = malloc(*nouvelle_taille * sizeof(entree));
  for (int j = 0 ; j < *nouvelle_taille ; j += 1)
  {
    reponse[j].cle = temporaire[j].cle;
    reponse[j].valeur = temporaire[j].valeur;
  }
  free(temporaire);
  return reponse;
}
```

Exercice 2 :

```
let randint a b = a + Random.int (b - a + 1);;

let choice_et_retrait l =
  let ind = Random.int (List.length l) in (* pas trop le choix pour length *)
  let rec aux i liste = match liste with
  | [] -> failwith "Cas impossible"
  | a::q -> if i = 0 then a, q else let x, reste = aux (i-1) q in x, a::reste
  in aux ind l

let choice l = fst (choice_et_retrait l)

let rec sample l n =
  if n = 0 then [] else let x, reste = choice_et_retrait l in x :: sample reste (n-1)
```

Exercice 3 :

Avec cette version naïve, la fonction `sample` fait `n` appels à la fonction `choice_et_retrait`, chacun de complexité linéaire en la taille de la liste. La complexité finale est le produit des deux valeurs. Une version plus rapide ferait un mélange en complexité linéaire en la taille de la liste (le plus simple mais un peu affreux est de passer par un tableau) puis récupérerait en temps linéaire en `n` les `n` premiers éléments. La complexité serait alors linéaire en la taille de la liste puisque cette valeur est censée être supérieure à `n`.

Exercice 4 :

```
let ri n =
  let entier = ref 0 in
  for i = 0 to 62 do entier := !entier * 2 + (if randbool () then 1 else 0) done;
  !entier mod n
```

Problème 1

Question P1.1 :

```
struct filelimitee { int* donnees ; int capacite ; int ecriture ; int lecture ; };
typedef struct filelimitee file_limitee;

file_limitee creer_fl(int capacite)
{
  file_limitee rep = { .donnees = malloc(capacite * sizeof(int)) ,
  .capacite = capacite , .ecriture = 0 , .lecture = 0 };
  return rep;
}

bool est_vide_fl(file_limitee f)
{
  return f.lecture == f.ecriture;
}

void enfiler_fl(file_limitee *f, int x)
{
  if (f->ecriture == f->capacite) exit(1);
  f->donnees[f->ecriture] = x;
  f->ecriture += 1;
}

int defiler_fl(file_limitee *f)
{
  if (f->lecture == f->ecriture) exit(1);
  f->lecture += 1;
  return f->donnees[f->lecture-1];
}
```

Question P1.2 :

```
void renverser(int pile p)
{
    f = creer_fl(taille_pile(p));
    while (!estvide(p))
    {
        enfiler_fl(&f, depiler(p));
    }
    while (!est_vide_fl(f))
    {
        empiler(p, defiler_fl(&f));
    }
} // Fuite de mémoire, mais j'ai oublié de demander une fonction de destruction dans P1.1.
```

Question P1.3 :

```
struct filebornee { int* donnees ; int capacite ; int taille ; int ecriture ; int lecture ; };
typedef struct filebornee file_bornee;

file_bornee creer_fb(int capacite)
{
    file_bornee rep = { .donnees = malloc(capacite * sizeof(int)) ,
        .capacite = capacite , .taille = 0 , .ecriture = 0 , .lecture = 0 };
    return rep;
}

bool est_vide_fb(file_bornee f)
{
    return f.taille == 0;
}

bool est_pleine_fb(file_bornee f)
{
    return f.taille == f.capacite;
}

void enfiler_fb(file_bornee *f, int x)
{
    if (f->taille == f->capacite) exit(1);
    f->donnees[f->ecriture] = x;
    f->taille += 1;
    f->ecriture = (f->ecriture + 1) % f->capacite;
}

int defiler_fb(file_bornee *f)
{
    if (f->taille == 0) exit(1);
    int rep = f->donnees[f->lecture];
    f->taille -= 1;
    f->lecture = (f->lecture + 1) % f->capacite;
    return rep;
}
```

Question P1.4 :

- On crée le tableau d'entiers correspondant aux caisses, indiquant le moment où la caisse est libre, et la file.
- Tant qu'au moins un client n'a pas quitté la file (en particulier s'il n'y est pas entré), on exécute le corps de la boucle principale : (1) Tant qu'il y a de la place dans la file et un client dans le tableau qui n'est pas encore passé par la file, on met un client dans la file. (2) On transfère un client à toutes les caisses libres à concurrence du nombre de clients dans la file d'attente. (On répète ces deux dernières étapes jusqu'à ce qu'il n'y ait plus rien à faire.) (3) On avance dans le temps jusqu'à la libération d'au moins une caisse (avec une astuce, la partie précédente permet d'obtenir cette information à la volée).
- On regarde à quel moment toutes les caisses sont libérées par un simple calcul de maximum.

```

int temps_supermarche(int* clients, int taille, int n, int k)
{
    int* caisses = malloc(n * sizeof(int));
    for (int i = 0 ; i < n ; i += 1) caisses[i] = 0;
    file_bornee f = creer_fb(k);
    int temps = 0;
    int ind_client = 0;
    while (ind_client < taille && !est_vide_fb(f))
    {
        int prochain = -1;
        bool evolution = true;
        while (evolution)
        {
            evolution = false;
            while (ind_client < taille && !est_pleine_fb(f))
            {
                enfiler_fb(&f, clients[ind_client]);
                ind_client += 1;
                evolution = true;
            }
            for (int i = 0 ; i < n ; i += 1)
            {
                if (caisses[i] <= temps && !est_vide_fb(f)) // en pratique ==
                {
                    caisses[i] += defiler_fb(&f);
                    if (prochain == -1 || caisses[i] < prochain) prochain = caisses[i];
                    evolution = true;
                }
            }
        }
        temps = prochain;
    }
    for (int i = 0 ; i < n ; i += 1)
    {
        if (caisses[i] > temps)
            temps = caisses[i];
    }
    return temps;
}

```

Question P1.5 :

```

struct maillon_file { int valeur; struct maillon_file* suivant; };
typedef struct maillon_file maillon;

struct file_finale { maillon* premier; maillon* dernier; };
typedef struct file_finale file;

file creer_file()
{
    file f = { .premier = NULL , .dernier = NULL };
    return f;
}

void enfiler(file* f, int x)
{
    maillon* d = f->dernier;
    maillon* mx = malloc(sizeof(maillon));
    mx->valeur = x;
    mx->suivant = NULL;
    d->suivant = mx;
    f->dernier = mx;
}

```

```

bool est_vide(file f)
{
    return f.premier == NULL;
}

int defiler(file* f)
{
    if (est_vide(*f)) exit(1);
    maillon* p = f->premier;
    int reponse = p->valeur;
    f->premier = p->suivant;
    free(p);
    return reponse;
}

```

Question P1.6 :

Pour ne pas laisser les files mutées après la fonction, on peut les transférer chacune dans une copie et retransférer.

```

file somme(file f1, file f2)
{
    file reponse = creer_file();
    int retenue = 0;
    while (!est_vide(f1) && !est_vide(f2))
    {
        int e1 = defiler(&f1);
        int e2 = defiler(&f2);
        int s = e1 + e2 + retenue;
        if (s >= 2)
        {
            s -= 2;
            retenue = 1;
        }
        else retenue = 0;
        enfiler(&reponse, s);
    }
    while (!est_vide(f1)) // rien ne se passe si f1 était vide
    {
        int e1 = defiler(&f1);
        int s = e1 + retenue;
        if (s >= 2)
        {
            s -= 2;
            retenue = 1;
        }
        else retenue = 0;
        enfiler(&reponse, s);
    }
    while (!est_vide(f2)) // analogue
    {
        int e2 = defiler(&f2);
        int s = e2 + retenue;
        if (s >= 2)
        {
            s -= 2;
            retenue = 1;
        }
        else retenue = 0;
        enfiler(&reponse, s);
    }
    if (retenue == 1) // à ne pas oublier
        enfiler(&reponse, 1);
    return reponse;
}

```

Problème 2

Question P2.1 :

$1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, $4 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$

On remarque que le passage de n à $n + 1$ ajoute un élément (qui n'est pas déjà dans l'ensemble) et que 0 est vide, donc le représentant de n est de taille n .

Question P2.2 :

On ne peut pas représenter les ensembles par de simples listes, car le typage de la fonction attendue poserait problème. Il s'agit donc d'inventer un type somme : `type vonneumann = Ensemble of vonneumann list.`¹

```
let rec vn n = if n = 0 then Ensemble []
else match vn (n-1) with Ensemble l -> Ensemble ((Ensemble l)::l)
```

On remarque une belle transcription des maths à OCaml : si $blabla \rightarrow blibli$, alors $\{blabla\} \rightarrow \text{Ensemble}(blibli)$.

Question P2.3 :

```
type entier_peano = Zero | S of entier_peano
```

Question P2.4 :

```
let entier_to_peano n =
  if n < 0 then failwith "Entier négatif, non représentable";
  let rec aux n = if n = 0 then Zero else S (aux (n-1))
  in aux n (* sous-fonction pour ne pas tester la négativité à chaque fois *)
```

```
let rec peano_to_entier e = match e with
| Zero -> 0
| S em1 -> 1 + peano_to_entier em1
```

Question P2.5 :

```
let rec somme e1 e2 = match e2 with
| Zero -> e1
| S(e2m1) -> somme (S e1) e2m1
```

Question P2.6 :

```
let rec difference e1 e2 = match e1, e2 with
| _, Zero -> e1
| Zero, _ -> failwith "Différence strictement négative"
| S(e1m1), S(e2m1) -> difference e1m1 e2m1
```

Question P2.7 :

```
let rec produit e1 e2 = match e2 with
| Zero -> Zero
| S(e2m1) -> somme e1 (produit e1 e2m1)
```

Question P2.8 :

```
let rec quotient e1 e2 =
  if e2 = Zero then failwith "Division par zéro";
  try let q1, r1 = quotient (difference e1 e2) e2 in S q1, r1
  with _ -> Zero, e1
```

1. Il est effectivement imaginable de créer un type avec comme sans constructeur constant. Cela peut troubler au vu de la définition d'un ensemble inductif, mais on fera parfois ainsi pour les arbres, et la fin des récursions se fait quand la liste est vide. Moralement, on crée un ensemble inductif avec un ensemble de base qui est `Ensemble []` et une règle de construction pour chaque arité entière strictement positive.