université PARIS-SACLAY

Introduction à la programmation fonctionnelle
L2 Informatique

semaine du 2 novembre 2020

TP n° 1

Consignes les exercices ou questions marqués d'un ⋆ devront être rédigés sur papier (afin de se préparer aux épreuves écrites de l'examen). En particulier, il est recommandé d'être dans les même conditions qu'en examen : pas de document ni de calculatrice. Tous les TPs se font sous Linux.

0 Prise en main de l'environnement

Pour faciliter les TPs en mode distanciel, nous utiliseront le serveur Jupiter hébergé par le LAL (Laboartoire Accélérateur Linéaire). Dans cet environnement, vous avez accès, via un navigateur Web à :

- un espace de stockage
- un terminal Unix
- un éditeur de texte avec la coloration syntaxique pour OCaml
- les commandes vues en cours : `ocamlc`, `ocaml`, ...

1. Se rendre sur `https://jupytercloud.lal.in2p3.fr/`
2. Cliquer sur se connecter
3. Utiliser l'authentification centralisée de Paris-Saclay (si vous êtes déjà connecté via eCampus, il est possible que l'authentification se fasse automatiquement)
4. Si vous êtes redirigé vers la page de garde : `https://jupytercloud.lal.in2p3.fr/`, c'est un bug connu. Effacez vos cookies ou ouvrez l'URL dans une fenêtre de navigation privée
5. Arrivé sur l'interface, choisissez « Start My Server » (bouton vert)

Vous vous retrouvez dans un navigateur de fichier. Vous pouvez créer des répertoires et des fichiers. Le bouton « New » à droite comporte plusieurs options dont :

- Create Terminal : ouvre un terminal dans un autre onglet de votre navigateur
- Create Texte File : ouvre un éditeur de texte

Dans l'éditeur de texte, si vous renommez le fichier en lui donnant l'extension « `.ml` » la coloration syntaxique pour OCaml est activée.

Il est **vivement recommandé** de télécharger vos fichiers après chaque exercice, pour en avoir une copie sur votre ordinateur personnel. Les sauvegardes sur cet espace de travail expérimental ne sont pas garanties, et c'est un espace distinct de vos répertoires personnels sur les machines de l'Université.

1 Lecture de code

⋆ Pour chacun des programmes OCaml suivants, dire ce qu'ils affichent.

(a)
```
1 let x = 3 ;;
2 let y = 4 ;;
3 let z = x + y ;;
4 Printf.printf "%d\n" z;;
```

(b)
```
1 let f x = (x + 1) * (x - 1)
2 ;;
3 let u = f (f 2) ;;
4 Printf.printf "%d\n" u;;
```

(c)
```
let x = 47 ;;
let y =
   if x mod 2 = 0 then
     3
   else
   4
;;
Printf.printf "%d\n" y;;
```

(d)
```
let f x = x / 17 ;;
let g x =
   if x mod 2 = 0 then
   "pair"
   else
   "impair"
   ;;
let u = f 42;;
let () =
   Printf.printf
   "%d est %s\n" u (g u);;
```

(e)
```

let x = 3.14159;;
let x2 = x. *. x. ;;
let x4 = x2 *. x2 ;;
Printf.printf "%f\n" x4;;
```

(f)
```
let rec f n =
   if n >= 0 then begin
       Printf.printf "%d\n" n;
       f (n-1)
    end
;;
let () = f 10;;
```

2 Entrées/Sorties

Écrire un programme qui attend en argument sur sa ligne de commande un nom de fichier, qui teste si ce fichier existe et qui teste son type. Plus précisément :

- S'il n'y a pas d'argument sur la ligne de commande, le programme affiche `"Pas assez d'arguments"`.
- S'il y a un argument mais qu'il ne correspond pas à un nom de fichier existant, le programme affiche `"Le fichier n'existe pas"`
- Si le fichier existe, le programme affiche `"Le fichier existe"` ou `"Le répertoire existe"` selon le cas.

Les fonctionalités utiles pour réaliser cet exercice sont :

- `Sys.argv` : le tableau des arguments passés au programme. Il contient toujours au moins une entrée : le nom du programme. Si vous utilisez l'interpréteur en ligne `tryocaml`, vous pouvez sauter cette fonctionnalité et définir directement le nom du fichier à tester dans une variable globale.
- `Array.length` : une fonction renvoyant la taille d'un tableau passé en argument.
- `Sys.file_exists` : une fonction qui teste si un nom de fichier existe.
- `Sys.is_directory` : une fonction qui teste si un nom de fichier est un répertoire.

Remarque si vous utilisez l'interpréteur en ligne `tryocaml`, les répertoires `/` et `/worker_cmis` existent. Le fichier `/worker_cmis/biggarray.cmi` existe.

3 Fonctions récursives

1. Écrire une fonction `somme_entiers n` qui renvoie la somme des entiers entre `n` et `0`. On pourra s'inspirer de la fonction (f) de l'exercice 1, en traitant le cas `else` (*i.e.* que faire lorsque `n` vaut 0).
2. Écrire une fonction `somme_carres n` qui renvoie la somme des i^2 pour i entre `0` (inclus) et `n` (exclu).
3. Écrire une fonction `leibniz n` qui calcule l'approximation de $\frac{\pi}{4}$ en utilisant la formule de Leibniz :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + (-1)^i \times \frac{1}{2i+1} + \ldots$$

4. Tester ces fonctions en les appelant sur différentes valeurs.

4 Dates

1. Écrire une fonction OCaml `bissextile a` qui renvoie `true` si `a` est bissexstile et `false` sinon. Quel est le type de cette fonction ?

2. Écrire une fonction `jour_mois m a` qui prend en argument un entier `m` compris entre 1 et 12 et qui renvoie le nombre du jour dans le mois `m` de l'année `a`.

   Remarque plutôt que d'utiliser 11 `if then else`, on peut remarquer que, si on exclu le mois de février alors :
   — les mois 8 à 12 se comportent comme les mois 1 à 8.
   — pour les mois entre 1 et 8, si le mois est impair il a 31 jour et s'il est pair il a 30 jours.

3. Écrire une fonction `jour_date j m a` qui prend trois arguments représentant un jour `j` (à partir de 1), un mois `m` (entre 1 et 12) et une année `a` et qui renvoie le nombre de jour écoulé entre cette date et le premier janvier de l'année.

   Remarque on pourra utiliser une fonction récursive pour parcourir les mois, en s'inspirant de l'exercice 3. En particulier :

   Cas de base : si le mois est janvier, il suffit renvoyer le nombre de jour de la date (le 4/1/2019 est bien le 4ème jour de l'année 2019).

   Cas récursif : pour une date `j m a`, le nombre de jours est la somme du nombre de jour dans le mois précédant `m` et (récursivement) le nombre de jour à la date du mois précédent : `j (m−1) a`.

4. Écrire un programme qui teste vos fonctions sur plusieurs dates.

5 Tours de Hanoï

Le problème des Tours de Hanoï est un célèbre problème possédant une élégante solution sous forme de fonction récursive. Le jeu consiste en trois piquets, A, B, C sur lesquels ont peut enfiler des disques de tailles décroissantes. Le but du jeu est de partir d'une pyramide de n disque sur le piquet A et de déplacer tous les disques vers le piquet C. Une contrainte est qu'il est interdit de placer un disque sur un disque plus petit. La figure 1 indique une configuration. Dans cette partie, il est interdit de déplacer le disque 4 sur le disque 1.

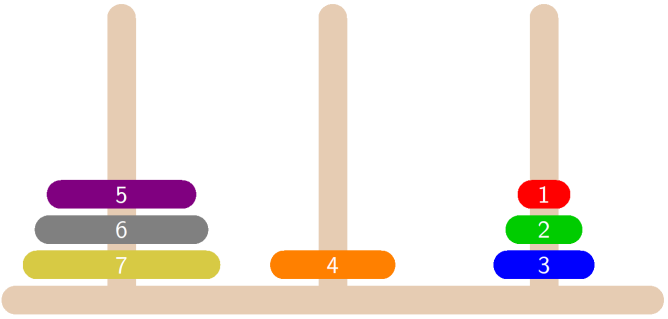

FIGURE 1 – Les tours de Hanoï

Pour déplacer le disque 4 sur le piquet de droite, on peut effectuer la séquence de coups suivante :
— déplacer 1 sur 5 (à gauche)
— déplacer 2 sur 4 (au milieu)
— déplacer 1 sur 2 (au milieu)
— déplacer 3 sur 5 (à gauche)
— déplacer 4 sur le piquet de droite.

1. On veut résoudre se problème à l'aide d'un programme, c'est à dire étant donné le nombre n de disque au départ, afficher la liste des mouvements à faire. Compléter le squelette de code proposé :

```
1  let rec hanoi dep aux dest n =
2    if n > 0 then begin
3    (* ...  *)
4    (* ...  *)
5    (* ...  *)
6    end
```

```
7
8   let () = hanoi "A" "B" "C" 5
```

La fonction affiche les coups à jouer sous la forme `A -> C` pour indiquer qu'il faut déplacer le disque se trouvant au sommet du piquet `A` vers le piquet `C`.

2. Transformer le code pour n'avoir qu'une seule fonction **`hanoi n`** qui appelle la fonction récursive auxiliaire **`hanoi_aux`** `"A" "B" "C"` **`n`**.
3. La fonction `Sys.time ()` renvoie un nombre flottant représentant le nombre de secondes écoulées depuis le début du programme. Modifier le programme pour mesure le temps pris par la fonction **`hanoi`** pour s'exécuter. À partir de quelle valeur de **`n`** dépasse t'on 10 secondes ?