université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Introduction à la programmation fonctionnelle
L2 Informatique

semaine du 16 septembre 2024

TP n° 2

0 Environnement de travail

0.1 Session de secours

Les sessions de secours étant recrées tous les jours, il est conseillé de travailler de la façon suivante :
— télécharger l'archive ZIP contenant les fichiers à compléter et la décompresser. Cette dernière contient un unique répertoire `ipf_l2_info_tpxx` où `xx` est le numéro du TP.
— ouvrir un terminal et se placer dans le répertoire en question `cd chemin/vers/le/repertoire`
— exécuter le script d'initialisation `./init_tp.sh`
— fermer le terminal et en ouvrir un nouveau
— travailler (voir la section 0.2)
— sauvegarder régulièrement et au moins une fois en fin de séance les fichiers du TP (soit en ligne sur un espace personnel [1], soit en les stockant sur une clé USB)

0.2 Utilisation de Visual Studio Code

Une fois configuré correctement (par le script d'initialisation), VSCode est un éditeur confortable pour du code OCaml (moins lourd que Netbeans ou Eclipse en particulier). Voici l'ensemble minimal des commandes pour les TPs :
— Création d'un nouveau Fichier : `CTRL-N` (ou « *File* → *New file* ». Attention, le fichier nouvellement créé n'a pas de nom. Il convient alors de l'enregistrer (`CTRL-S`) ou *Save...*) en lui donnant un nom se terminant par `.ml`
— Évaluation d'une expression OCaml : il est possible de surligner (SHIFT+↑ / ↓ ou en utilisant la souris) une portion de code OCaml puis de l'envoyer dans un interpréteur avec `SHIFT-ENTER`. Attention, il convient d'évaluer les définitions dans l'ordre.

1 Lecture de code

⋆ Dire si les programmes suivants sont bien typés ou mal-typés. S'ils sont bien typés, donner le type de toutes les variables et fonctions globales du fichier. S'ils sont mal typés, indiquer précisément la ligne et la nature de l'erreur de typage.

(a)
```
1 let x = 3
2 let y = 4
3 let z =
4     string_of_int x + y
```

(b)
```
1 let f x y = (x + 1) * (y - 1)
2 let u = f (f 2 3) (f 4 5)
```

1. a priori seul le HTTP et HTTPS sortant est autorisé à l'heure actuelle, il est donc possible que l'utilisation de `git` via SSH ne fonctionne pas

(c)
```
let x = 47
let y =
   if x mod 2 = 0 then
     3.0
   else
   4.0
let z = x + y
```

(d)
```
let f x = x / 17
let g x =
   if x mod 2 = 0 then
   "pair"
   else
   false

let u = f 42
```

(e)
```
let rec f n =
    if n <= 0 then 1
    else n * f (n-1)

let g n =
  Printf.printf
    "%d! = %d\n" n (f n)
```

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

2 Trouve le nombre

Le petit jeu « trouve le nombre » (dans lequel l'ordinateur choisit aléatoirement un nombre entre 0 et 100 et le joueur doit le deviner en le moins d'étapes possibles) est typiquement montré dans les premiers cours de programmation impérative comme exemple d'utilisation de la boucle `while`. On propose d'illustrer le fait que « tout ce qui peut être fait avec une boucle peut aussi l'être avec une fonction récursive ».

1. écrire une fonction `guess : int -> unit` prenant en argument un entier `n`. La fonction lit un entier au clavier au moyen de `read_int ()` puis compare cet entier à `n`. Si les deux sont égaux, la fonction affiche `"Trouvé !"`. Sinon elle affiche `"Trop petit!"` ou `"Trop grand!"` puis se rappelle récursivement
2. appeler la fonction sur un entier `n` choisit aléatoirement. On pourra utiliser les fonctions :
   — `Random.self_init : unit -> unit` qui initialise le générateur de nombres aléatoires
   — `Random.int : int -> int` qui attend un argument entier `n` positif et renvoie un nombre aléatoire entre 0 et `n-1` inclus.
3. modifier la fonction `guess` pour qu'elle affiche en plus le nombre d'essais effectués. **Indication** la fonction devient `guess : int -> int -> unit` le second argument étant le nombre d'essais faits jusqu'à présent.

3 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 disques 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. 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 disques au départ, afficher la liste des mouvements à faire. Compléter le squelette de code proposé :

```
let rec hanoi dep aux dest n =
  if n > 0 then begin
```

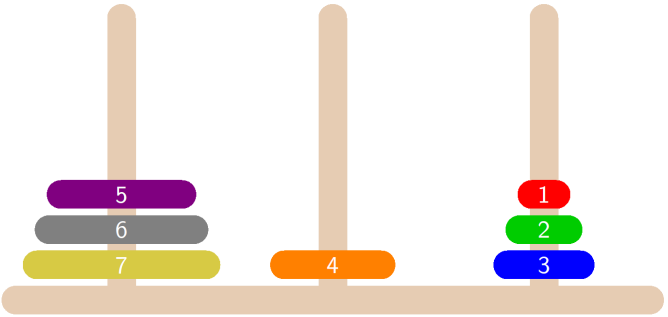

FIGURE 1 – Les tours de Hanoï

```
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 mesurer le temps pris par la fonction `hanoi` pour s'exécuter. À partir de quelle valeur de `n` dépasse-t-on 10 secondes ?

4 Parcours d'un répertoire

Le but de l'exercice est de s'inspirer du programme 2 de la feuille de TP 1 et de le complexifier pour qu'il reproduise le comportement de la commande Unix : `ls -R d`. Cette dernière affiche récursivement le contenu d'un répertoire et de tous les répertoires contenus dans ce dernier.

1. Écrire une fonction `affiche_tab a` qui prend en argument un tableau de chaînes de caractères `a` et qui affiche le contenu des cases, chacun sur une ligne avec un retour à la ligne et décalé de 4 espaces par rapport au début de la ligne. On pourra utiliser une fonction récursive `affiche_tab_aux i a` qui prend en argument l'indice courant.
2. Écrire du code de test qui appelle `affiche_tab Sys.argv` et vérifier que tous les arguments de votre ligne de commande sont bien affichés.
3. Créer deux fonctions mutuellement récursives :

```
1 let rec affiche_dir d =
2   ...
3 and affiche_dir_iter i t =
4   ...
```

où `affiche_dir d` :

cas de base : ne fait rien si `d` n'est pas un répertoire (tester avec `Sys.is_directory`)

cas récursif :

— Affiche `d` en début de ligne, suivi de « : » et un retour à la ligne.

— Récupère le contenu du répertoire `d` au moyen de `Sys.readdir`. Cette fonction prend en argument le nom d'un répertoire et renvoie son contenu sous la forme d'un tableau t.

— Affiche t au moyen de `affiche_tab`
— Fait de `d` le répertoire courant (en utilisant `Sys.chdir`)
— appelle récursivement `affiche_dir_iter 0 t`.
— Remonte dans le répertoire initial avec `Sys.chdir ".."`

La fonction affiche_dir_iter i t appelle récursivement `affiche_dir t.(`j`)` pour toutes les indices j compris entre `i` et la taille de `t`.

Enfin, faire en sorte que votre programme appelle `afffiche_dir "."` (i.e. affiche récursivement le répertoire courant).

4. Créer (dans le terminal, pas en OCaml) un ensemble de répertoires :

```
$ mkdir -p a/b/c
$ mkdir -p a/d/e
$ touch a/f1.txt a/b/f2.txt a/d/e/f3.txt
```

Tester votre programme en l'exécutant dans le répertoire courant. Il doit afficher tous les fichiers s'y trouvant ainsi que les répertoires `a`, `b`, `c`, `d`, `e` et leur contenu.

5. Pour chacune des fonctions, `affiche_tab_aux`, `affiche_dir`, `affiche_dir_iter`, dire si elle est récursive terminale ou non en justifiant.