

TP n° 3

Consignes les exercices ou questions marqués d'un \star devront être rédigés sur papier (afin de se préparer aux épreuves écrites de l'examen). En particulier, il est recommandé d'être dans les mêmes 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 utiliserons le serveur Jupiter hébergé par le LAL (Laboratoire 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

\star 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 f x = x+10, x-10
2 ;;
3 let a, b = f 42 42
4 ;;
```

(b)

```
1 type t = A of int | B of bool
2
3 let f x = match x with
4     A i -> A (i + 1)
5     | B b -> B (not b)
6     ;;
```

```

1 type t = A of int | B of bool
2
3 let f x = match x with
4     A i -> i + 1
5     | B b -> not b
6     ;;

```

```

1 let f x = (x, x);;
2 let g x = snd x;;
3 let u = g (f 42);;

```

```

1 let rec f op init n =
2     if n <= 0 then init
3     else op n (f op init (n-1))
4     ;;
5 let x = f ( * ) 1 10;;
6 let add_str n s =
7     (string_of_int s) ^ " " ^ s
8 let y = f add_str "" 9
9     ;;

```

2 Bibliothèques sur les nombres

Le but de cet exercice est d'écrire une petite bibliothèque permettant de manipuler des nombres. Les nombres peuvent être représentés, en interne de trois façons différentes :

- Comme des entiers
- Comme des flottants
- Comme des fractions

On souhaite pouvoir additionner, soustraire, multiplier ou diviser arbitrairement deux nombres, quel que soit le format interne. Les opérations doivent automatiquement faire la conversion vers la représentation interne la plus appropriée.

2.1 Fractions

Nous allons commencer par définir un type auxiliaire pour représenter des fractions. Une fraction $\frac{a}{b}$ est représentée par deux entiers, le numérateur a et le dénominateur b . On souhaite de plus avoir les contraintes suivantes :

- la fraction est irréductible, *i.e.* le PGCD de a et b vaut 1
- b est toujours positif, autrement dit, le signe de la fraction est donné par le signe de l'entier a

1. Écrire une fonction récursive **pgcd a b** qui calcule le PGCD de deux entiers, en utilisant l'algorithme (récurif) d'Euclide :

- si **b** est nul, renvoyer **a**
- sinon renvoyer le PGCD de **b** et **a mod b**

La fonction est-elle récursive terminale ?

2. Définir un type enregistrement **frac** possédant deux champs **num** et **denom**

3. Définir une fonction auxiliaire **sign i** qui renvoie 1, -1 ou 0 selon que **i** est positif, négatif ou nul.

4. Définir une fonction **simp f** qui simplifie la fraction **f** et s'assurer que **b** est positif.

5. Définir les fonctions suivantes :

frac a b : renvoie la fraction ayant pour numérateur **a** et pour dénominateur **b**

add_frac f1 f2 : additionne les deux fractions

neg_frac f : renvoie l'opposé de **f**

sub_frac f1 f2 : soustrait les deux fractions

mul_frac f1 f2 : multiplie les deux fractions

inv_frac f : renvoie l'inverse de **f**

div_frac f1 f2 : divise les deux fractions

string_of_frac f : convertit la fraction **f** en chaîne de caractères

float_of_frac f : convertit la fraction en nombre flottant

Toutes les fonctions qui renvoient des fractions doivent renvoyer des fractions simplifiées. On évitera un maximum de dupliquer du code, en utilisant les fonctions précédemment définies le plus possible. On testera aussi ces fonctions.

2.2 Nombres

Un *nombre* peut être représenté de trois façons différentes. Nous allons pour cela utiliser un type somme OCaml :

```
1  type num =
2    Int of int
3    | Float of float
4    | Frac of frac
5  ;;
```

1. Écrire une fonction `string_of_num n` qui renvoie une chaîne de caractères représentant le nombre `n` donné en argument. On pourra utiliser soit les fonctions prédéfinies `string_of_int` et `string_of_float` soit la fonction `Printf.sprintf "..."`... qui fonctionne comme `Printf.print` mais renvoie la chaîne formatée plutôt que de l'afficher dans la console.

On souhaite maintenant écrire des opérations génériques entre valeur du type `num`. Considérons une expression $n_1 \square n_2$ (où \square représente l'addition, la soustraction, la multiplication ou la division).

- Si n_1 ou n_2 sont de même type, alors le résultat de $n_1 \square n_2$ est de ce type
- Si n_1 ou n_2 est un flottant, alors l'autre opérande est convertie en flottant et le résultat est un flottant
- Sinon si n_1 ou n_2 est une fraction, alors l'autre opérande est convertie en fraction et le résultat est une fraction

Afin de réutiliser un maximum le code, on veut écrire une fonction générique `exec_op n1 n2 op_i op_fr op_fl` prenant en argument deux nombres et trois fonctions :

`op_i : int -> int -> int` est une opération entre entiers

`op_fr : frac -> frac -> frac` est une opération entre fractions

`op_fl : float -> float -> float` est une opération entre flottants

```
1  let exec_op n1 n2 op_i op_fr op_fl =
2  match n1, n2 with
3  | Float f1, Float f2 -> Float (op_fl f1 f2) (* deux flottants *)
4  | Float f1, Frac fr2 -> Float ( ... )      (* flottant et fraction *)
5  | ...
6  ...
7  ;;
```

Le type de la fonction ci-dessus est donc :

2. Donner le type de la fonction `exec_op`
3. Compléter la fonction `exec_op`.
4. Définir les fonctions :
 - `add_num n1 n2` : additionne les deux nombres
 - `sub_num n1 n2` : soustrait les deux nombres
 - `mul_num n1 n2` : multiplie les deux nombres
 - `div_num n1 n2` : divise les deux nombres
5. Définir une fonction récursive terminale `pow n k` prenant en argument un nombre `n` et un entier (`type int`) `k` et qui calcule n^k .