université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

semaine du 2 septembre 2024

Programmation Fonctionnelle Avancée
L3 Info – LDD3 Mag. Info

TP n° 1

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 `pfa_l3_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.

De façon générale, les archives de TP contiennent déjà un ou plusieurs fichier à compléter, il est rare de devoir créer des fichiers.

Objectif

Cette feuille de TP est *volontairement* longue afin de nous permettre d'évaluer votre niveau général en OCaml. Elle n'est *pas* notée (évidemment !), et le corrigé sera fourni après la séance (comme toujours). Il est donc important, dans la mesure du possible, de travailler seul en séance de TP et non pas à deux (un ou une qui est au clavier et l'autre qui « réfléchit »). Si à la suite de cette séance vous constatez des difficultés particulières, mentionnez les en amphi quelques minutes y seront consacrées au début du cours 2.

1 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

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

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.

1.1 Fractions

Nous allons commencer par définir un type auxiliaire pour représenter des fractions. Un 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 (récursif) d'Euclide :
   — si `b` est nul, renvoyer `a`
   — sinon renvoyer le PGCD de `b` et `a mod b`
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'assure que `b` est positif.
5. Définir les fonctions suivantes :

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

   `string_of_frac f` : convertit la fraction `f` en chaîne de caractères
6. Définir du code de test dans votre fichier. Par exemple :

```
1   let pr_frac a b =
2     Prinf.printf "%d / %d : %s\n" a b (string_of_frac (frac a b))
3
4   (* tests simples *)
5   let test1 () =
6     pr_frac 1 2;
7     pr_frac 2 4;
8     pr_frac 10 25
9
10  (* tests sur des négatifs *)
11  let test_2 () =
12    pr_frac (-1) 2;
13    pr_frac (-2) (-10);
14    pr_frac 3  (-6)
15
16  ...
17
18  let () =
19    test_1 ();
20    test_2 ()
```

   Vous pouvez ensuite tester le programme depuis le terminal avec :

```
$ ocamlc -o nombre.exe nombre.ml && ./nombre.exe
```

7. Écrire les fonctions suivantes

   `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

`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. Ajouter des tests pour ces fonctions.

8. On se donne les deux fonction suivantes :

```
1  let rec fof_aux f =
2  let r, i = modf f in
3  let fi = (frac (int_of_float i) 1) in
4  if r  < 0.001 then fi
5  else
6    add_frac fi
7    (inv_frac (fof_aux (1.0 /.r)))
8
9  let frac_of_float f =
10   let s = if f < 0.0 then -1 else 1 in
11   mul_frac (frac s 1) (fof_aux (abs_float f))
```

Recopier ces fonctions et les tester. Que renvoient-elles ? (vous pouvez essayer de le déduire part leur type et en testant les résultats sans comprendre exactement ce que fait le code).

1.2 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
```

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 formattée plutôt que le l'afficher dans la console.

On souhaite maintenant écrire des opérations génériques entre valeurs 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 fl1, Float fl2 -> Float (op_fl f1 f2) (* deux flottants *)
4  | Float fl1, Frac fr2 -> Float ( ... )        (* flottant et fraction *)
5  | ...
6  ...
```

2. Compléter la fonction `exec_op`.
3. Définir les fonctions :

`add_num n1 n2` : additionne les deux nombres

`sub_num n1 n2` : soustrait les deux nombres

`mul_num n1 n2` : multiple les deux nombres

`div_num n1 n2` : divise les deux nombres

1.3 Calculatrice

On souhaite maintenant écrire une petite calculatrice effectuant des opérations entre nombres. Le fonctionnement général du programme est le suivant :

— saisie sur l'entrée standard d'un premier nombre n_1 (suivi de return)

— saisie sur l'entrée standard d'une opération o parmi `+`, `-`, `*`, `/` (suivie de return)

— saisie sur l'entrée standard d'un second nombre n_2 (suivi de return)

Affichage du résultat de $n_1 o n_2$ à la fois comme un entier, une fraction et un flottant, puis saisie d'un nouveau nombre n_1, puis o, puis n_2, *etc.*

1. Écrire une fonction `all_of_num n` qui étant donné une valeur `n` de type `num`, renvoie le triplet d'un entier, une fraction et un flottant le représentant. Si le nombre n'est pas représentable comme un entier, l'entier le plus proche du flottant ou de la fraction le représentant est utilisé. Si la conversion du nombre en fraction n'est pas possible, le comportement n'est pas spécifié (autrement dit, on utilisera `frac_of_float` si le nombre à convertir est un `Float f`).
2. Écrire une fonction `num_of_string s` qui convertit une chaîne de caractère en `num` en appliquant l'algorithme suivant. On utilise la fonction `String.index_opt s c` qui renvoie soit `Some i` où `i` est la position de la première occurrence du caractère `c` dans la chaîne `s` et `None` si ce caractère n'est pas présent.
   — Si le caractère `.` est présent dans la chaîne, appeler `float_of_string` sur cette dernière pour construire un flottant
   — Sinon si le caractère `/` est présent, découper la chaîne `s` en deux sous chaînes `s1` et `s2` telles que `s` $= s1\text{"}/\text{"}s2$ au moyen de la fonction `String.sub`. Cette dernière est telle que `String.sub s i l` renvoie la sous-chaîne de longueur l de `s` commençant à l'indice `i`. Convertir les deux sous-chaînes `s1` et `s2` en entier et former une fraction avec
   — Sinon la chaîne `s` est convertie en entier au moyen de `int_of_string`
3. Écrire la fonction suivante qui rattrape les exceptions éventuellement lancée par les fonctions de conversion de chaînes et renvoi un type option à la place :

```
1 let num_of_string_opt s =
2     try Some (num_of_string s) with _ -> None
```

   On rappelle que `Some ...` et `None` sont les constructeurs du type prédéfini `option` qui permet de représenter soit une valeur (`Some v`) soit une absence de valeur (`None`). La construction `try ... with` permet de rattraper des exceptions et sera rappelée en détail au cours 2.
4. Écrire une fonction `op_of_string_opt s` qui renvoie `Some f` où `f` est la fonction `add_num` si `s` vaut `"+"`, `sub_num` si `s` vaut `"-"`, `mul_num` si `s` vaut `"*"`, `div_num` si `s` vaut `"/"` et `None` dans les autre cas.
5. Écrire une fonction récursive `read_until f` qui :
   — lit une chaîne `s` sur l'entrée standard au moyen de `read_line()`
   — appelle `f s`
   — si `f s` renvoie `Some v`, renvoie `v`
   — sinon (si `f s` renvoie `None`) recommence à lire
6. Utiliser la fonction précédente pour implémenter la calculatrice. Cette dernière sera une fonction récursive *sans cas de base.*
7. Tester le programme (on pensera à commenter les tests des parties précédentes pour ne pas polluer l'affichage). On peut interrompre le programme avec (Ctrl-C) ?