université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

semaine du 25 septembre 2023

Introduction à la programmation fonctionnelle
L2 Informatique

TP n° 3

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.

1 Lecture de code

⋆ Dire si les programmes suivants sont bien typées ou mal-typées. S'ils sont bien typés, donner le type de toutes le variables et fonctions globales du fichier. S'ils sont mal typées, 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
```

(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)
```

(c)
```
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
```

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

(e)
```
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
```

Réponse:

a. Bien typé : `a : int`, `b : int`, `f : int -> (int * int)` .

b. Bien typé : `f: t -> t`. `f : int -> int -> int`, `u : int`.

c. Mal typé. Une branche est de type `int` l'autre est de type `bool`. `x + y` sans conversion, erreur de typage ligne 7.

d. Bien typé : `f : 'a -> 'a * 'a`, `g : ('a * 'b) -> 'b`, `u : int`. Attention, pour `g` les éléments de la paire n'ont a priori pas le même type.

e. Bien typé :
- `f` attend 3 arguments elle est de type : $T_{\texttt{op}} -> T_{\texttt{init}} -> T_{\texttt{n}} -> T_{\texttt{f}}$.
- $T_{\texttt{n}}$ vaut `int` car on compare à 0 et on fait `n - 1`.
- $T_{\texttt{op}}$: $T_{\texttt{a}} -> T_{\texttt{b}} -> T_{\texttt{c}}$, en effet, avec la ligne 3 on peut constater que `op` est une fonction. De plus on l'applique à `n` dont $T_{\texttt{a}} = \texttt{int}$.
- Le résultat de `op` est utilisé comme réulstat pour `f`, on a donc : $T_{\texttt{c}} = T_{\texttt{f}}$.

— `init` est utilisé comme résultat pour `f`, on a donc : $T_{\texttt{init}} = T_{\texttt{f}}$.
Type final :

```
(int -> 'a -> 'a) -> 'a -> int -> 'a
```

On a ensuite :`x : int` (la fonction calcule factorielle 10), `add_str : int -> string -> int` et `y : string` (la valeur de y est `"9 8 7 6 5 4 3 2 1 "`).

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. 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`
   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'assure 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 :

```
type num =
  Int of int
| Float of float
| 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

```
let exec_op n1 n2 op_i op_fr op_fl =
match n1, n2 with
| Float fl1, Float fl2 -> Float (op_fl f1 f2) (* deux flottants *)
| Float fl1, Frac fr2 -> Float ( ... )        (* flottant et fraction *)
| ...
...
```

2. Donner le type de la fonction `exec_op`

Réponse:

```
num -> num ->
    (int -> int -> int) -> (frac -> frac -> frac) -> (float -> float -> float) -> num
```

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` : multiple 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.