# TD 2 : Typage de programmes fonctionnels

## 1 Implantation de règles de typage

1. Concevez en Caml des types de données `expr` (pour des expressions) et `stmt` (pour des instructions/ *statement*) qui permettent de représenter le langage impératif donné par la grammaire p. 24 des transparents du cours.
2. Représentez les exemples vus en cours comme instances de ces types de données.
3. Écrivez des fonctions qui permettent d'imprimer des expressions / instructions.
4. Concevez un type de données `tp` pour représenter les types élémentaires (p. 22 des transparents).
5. Concevez une structure de données pour représenter des environnements. Implantez les algorithmes "essentiels" pour la manipulation de cette structure de données (ajout d'une déclaration, recherche d'un type associé au nom d'une variable).
6. Écrivez les fonctions `verif_tp_expr` et `verif_tp_stmt` qui implantent les règles de typage pour expressions et instructions. En cas d'échec de la vérification, ces fonctions doivent fournir un message d'erreur compréhensible.

## 2 Règles de typage : extensions

Vous avez vu en TP comment implanter des règles de typage en Caml. Étendez cette implantation, en

- rajoutant d'autres opérateurs binaires au type de données (arithmétiques : soustraction, multiplication ; de comparaison : égal et inférieur, logiques : et, ou). Comment éviter une duplication innécessaire de code ?
- implantant les règles de typage pour ces opérateurs
- générant des messages d'erreur pertinents, le cas échéant (par exemple : donner le nom d'une variable non déclarée, imprimer la sous-expression mal typée etc.)

## 3 Vérification de typage de programmes fonctionnels

Faites la vérification du typage pour les expressions suivantes sous l'environnement `Env = [(b, bool); (x, int); (f, int -> int -> int); (p, int -> bool)]` en supposant le typage traditionnel des constantes et des opérateurs arithmétiques et booléens :

Vous pouvez vous assurer d'avoir trouvé le bon type en faisant la vérification des expressions sous Caml, comme vu en cours.

- `(f x 3)`
- `(f x)`
- `fun (y : int) (f x y)`
- `fun (y : int) (f x)`
- `fun (y : int) (p y)`
- `fun (y : int) (p b)`
- `(p 3) = b`

Quelle est la réaction de l'interpréteur Caml si vous essayez de comparer deux fonctions, par exemple : `(fun x -> x) = (fun y -> y);;`