

TP n°1

Modules

Les exercices 1,2,3,6 de ce TP sont extraits du cours de Didier Rémy disponible à l'adresse <http://gallium.inria.fr/~remy/isia>.

1 Conversions entre monnaies

Exercice 1 [Monnaies] L'Euro et le Dollar sont deux monnaies supportant les mêmes opérations mais incompatibles entre elles.

1. Définissez un module `Mfloat` contenant un type `t = float`, ainsi que les opérations `un : t`, `plus : t -> t -> t`, et `prod : float -> t -> t` permettant de créer l'unité pour le type `t`, d'additionner deux éléments de type `t`, et de multiplier un élément par un scalaire.
2. Écrivez une interface `MONNAIE` de telle façon à ce que le type `t` soit abstrait.
3. On peut maintenant créer deux modules `Euro` et `Dollar` par le code suivant :

```
module Euro = (MFloat : MONNAIE);;  
module Dollar = (MFloat : MONNAIE);;
```

Vérifiez que les deux monnaies sont bien incompatibles entre elles. Quel mécanisme de OCaml permet de s'en assurer ?

4. Ecrivez une fonction `euro : float -> Euro.t` qui fabrique des Euros.

Exercice 2 [Bureau de change] On souhaite maintenant créer un bureau de change sous forme de module qui permette de convertir une somme en euros en une somme en dollars.

1. Donnez la signature du module la plus abstraite (par exemple le taux de change n'a pas besoin d'être connu). Rappel : la signature d'un module peut elle même contenir des variables de type module.
2. Donnez en une implémentation.

Exercice 3 [Restrictions] On souhaite définir un module qui permette uniquement d'additionner des euros mais qu'il soit en revanche impossible d'en créer ou d'en multiplier.

1. Donnez une signature `MPLUS` qui permette cette restriction.
2. A partir de cette signature et du module `Euro` créé précédemment, créez un module `Mplus` de la façon la plus concise possible.
3. Vérifiez que votre code fonctionne i.e. :
 - (a) L'expression suivante doit être bien typée :

```
MPlus.plus Euro.un Euro.un;;
```
 - (b) Mais l'expression suivante doit retourner une erreur :

```
MPlus.plus Euro.un 1.0;;
```

2 Compilation qui n'est plus séparée

Exercice 4 Reprenez le dernier exercice du TP précédent, et construisez un fichier source unique contenant des définitions de modules et interfaces de modules correspondantes aux fichiers `plus.ml`, `fois.ml`, `exp.ml` et `main.ml`. Attention, les fonctions doivent toujours être appelées `Plus.plus`, `Fois.fois` et `Exp.exponentielle`.

3 Intermède informatique

Exercice 5 On définit le module `Ordi` de la façon suivante :

```
module Ordi =
struct
  type t = int ref
  let create () = ref 0
  let start s = s:=1
  let read_state s = !s
end
```

Donnez deux signatures `USER_ORDI` et `ADMIN_ORDI` de telle façon à ce que le type `t` soit abstrait, et que seul l'administrateur puisse accéder aux fonctions `start` et `create`.

L'utilisateur pourra seulement utiliser la fonction `read_state`.

Exercice 6 Voici un extrait du code de `Menhir`, un générateur de parseurs pour OCaml :

```
(* A uniform interface for output channels. *)
module type OUTPUT = sig
  type channel
  val char: channel -> char -> unit (* Ecrit un caractère sur le canal *)
  val substring: channel -> string -> int (* offset *) -> int (* length *) -> unit
  (* Ecrit length caractères de la chaine string à partir de l'offset
  sur le canal channel *)
end
```

On souhaite donner deux implémentations de cette interface : une fonctionnant avec l'entrée standard, l'autre avec le module `Buffer`. A partir de la documentation du module `Buffer` et du module `Pervasives` (le module chargé par défaut qui contient les informations sur les entrées/sorties standards), écrivez `ChannelOutput` qui a comme type concret la sortie standard, et `BufferOutput` qui a comme type concret le type abstrait des buffers.

4 Banques

Exercice 7 1. Donnez deux signatures `Client` et `Banque` de module qui représentent une banque. L'une est destinée au client et comprendra les fonctions `depot` et `retrait` toutes deux de type `t -> monnaie -> monnaie`. Les types `t` et `monnaie` seront abstraits. L'autre signature est destinée au banquier et aura en plus de celle du client la possibilité de créer une Banque.

2. Donnez une implémentation du module `Banque` de type :

```

module Banque : functor (M : MONNAIE) ->
sig
  type t
  type monnaie = M.t
  val creer : unit -> t
  val depot : t -> monnaie -> monnaie
  val retrait : t -> monnaie -> monnaie
end

```

Vous donnerez un type concret pour `t`.

3. Créez une banque `Toto` tenant ses comptes en euros.
4. Donnez une implémentation de la signature `Client` qui soit compatible avec la banque `Toto` précédemment créée i.e. on peut faire :

```

let mon_ccp = Toto.creer ();;
Toto.depot mon_ccp (euro 100.0);;
Client.depot mon_ccp (euro 100.0);;

```

5. Créez une banque gérant les comptes en dollars et vérifiez qu'il n'est pas possible d'ajouter des euros.

5 Plus de foncteurs

Exercice 8 [Utilisation de foncteurs] Récupérez le code à l'adresse http://gallium.inria.fr/~scherer/teaching/fpav/2013/tp1_code.ml ou en local à `~scherer/sujets/tp1_code.ml`.

1. Grâce au foncteur `Set`, créez un module représentant les ensembles de chaînes de caractères.
2. Essayez de créer un ensemble contenant les éléments `"abc"`, `"def"`, `"ghi"`. Quel est le problème?
3. Après avoir résolu le problème, créez l'ensemble demandé à la question précédente et vérifiez le résultat de la fonction `mem` sur différentes valeurs.
4. Réécrivez le module `Set` en utilisant la notation simplifiée pour les foncteurs.

6 Foncteurs de la bibliothèque standard

Exercice 9 [Les foncteurs `Set` et `Hashtbl`]

- Ecrivez un module `Tree` représentant des arbres binaires d'entiers. Créez ensuite une table de hachage avec pour éléments ces arbres en utilisant le foncteur `Hashtbl.Make`. Pour la fonction de hachage, vous pourrez utiliser la fonction polymorphe du module `Hash`.
- Créez également un module contenant un ensemble d'arbres binaires ordonnées par leurs tailles grâce au foncteur `Set.Make` de la librairie standard (attention aux problème de visibilité si vous êtes dans le même environnement que l'exercice précédent!)
- Créez maintenant une table de hachage avec pour éléments des ensembles d'arbres (il peut être utile de recourir à l'instruction `include`).
- Testez votre code en créant des ensembles d'arbres binaires.