info401 : Programmation fonctionnelle
TP 5 : rendre de la monnaie, références et modules

Pierre Hyvernat
Laboratoire de mathématiques de l'université de Savoie
bâtiment Chablais, bureau 22, poste : 94 22
email : Pierre.Hyvernat@univ-savoie.fr
www : http://www.lama.univ-savoie.fr/~hyvernat/
wiki : http://www.lama.univ-savoie.fr/wiki

Ce TP, comme les suivants, sera noté. Je ne demande aucun compte rendu à part, mais seulement un *unique* fichier Caml, commenté.

- Votre fichier doit contenir du code valide et ne doit pas provoquer d'erreur lorsqu'on l'évalue avec l'interprète Caml. (Testez avant de me l'envoyer : si votre fichier ne s'évalue pas correctement, vous perdez automatiquement 5 points sur votre note finale.)
- Votre fichier devra contenir un commentaire contenant votre nom, prénom et filière. (Idem, si ce n'est pas le cas, vous perdez 5 points sur votre note finale.)
- Pour m'envoyer votre TP, utilisez uniquement le formulaire dont l'adresse est : (lien disponible sur le wiki)
  `http://www.lama.univ-savoie.fr/~hyvernat/envoi-TP.php`

Partie 1 : comment faire de la monnaie

La boulangère dispose d'une quantité illimitée de pièces de 1, 2, 5 et 10 centimes. Elle peut obtenir n'importe quelle somme à partir de ces pièces, mais comment faire pour minimiser le nombre total de pièces utilisées ?

Question 1. (Préliminaire) Proposez un algorithme simple pour trouver une solution optimale. Programmez une fonction correspondante :

```
monnaie_glouton : int -> (int*int) list
```

devra prendre en argument la somme S voulue et devra renvoyer une solution qui minimise le nombre total de pièces utilisées.

Par exemple : `monnaie_glouton 9` donnera `[ (5,1) ; (2,2) ]` car il suffit d'une pièce de 5 centimes et de 2 pièces de 2 centimes pour faire 9 centimes.

Question 2. Adaptez votre algorithme pour qu'on puisse lui donner en argument la liste des pièces de monnaies existantes :

```
monnaie_glouton : int list -> int -> (int*int) list
```

L'évaluation de l'expression `monnaie_glouton [1 ; 5 ; 25 ; 125] 142` devra donc donner le résultat `[ (1,125) ; (3,5) ; (2,1) ]`...

Question 3. Sur la planète kcahteN, les habitants utilisent des pièces de 1, 4 et 6 dimkroz. Que pensez-vous de l'algorithme précédent sur la planète kcahteN pour les valeurs entre 1 et 10 dimkroz ?

Question 4. Pour corriger le problème de la question précédente, nous allons utiliser un algorithme récursif plus complexe :

```
monnaie_rec 0 _ = 0
monnaie_rec _ [] = IMPOSSIBLE   (* ou bien : "+infini" *)
monnaie_rec n (p::l) = min (monnaie_rec n l)
                           (1 + (monnaie|rec (n-p) (p::l)))
```

Le premier cas correspond au cas où on n'utilise pas la pièce `p`, et le second au cas où on utilise la pièce `p`.

Écrivez la fonction `monnaie_rec` correspondante.

Question 5. Généralisez ce principe pour obtenir le nombre de chacune des pièces, et écrivez la fonction correspondante :

```
monnaie_rec : int list -> int -> (int*int) list
```

Question 6. Pour mémoizer la fonction, nous allons utiliser un petit module de type

```
module type TypeDict = sig
  type key = int*(int list)
  type (+'a) t                          (* type des dictionnaires *)
  val empty : 'a t
  val find : key -> 'a t -> 'a          (* pour trouver une valeur *)
  val add : key -> 'a -> 'a t -> 'a t (* pour ajouter une valeur *)
end
```

La fonction `find` devra lever l'exception `Not_found` quand l'élément n'existe pas dans le dictionnaire.

Créez un module `Dict` de ce type qui utilise les listes d'associations comme type des dictionnaires :

```
module Dict : TypeDict = struct
  type key = int*int
  type 'a t = (key*'a) list
  let empty = ...
  ...
end
```

Question 7. En utilisant une référence vers une valeur de type "`... Dict.t`" (dictionnaire avec des valeurs bien choisies), programmez une version mémoizée `monnaie_memoizee : int list -> int -> (int*int) list`* de la fonction précédente.

Consignes :

- réfléchissez avant de programmer,
- justifiez vos choix,
- pour garantir la transparence référentielle à l'extérieur de votre programme, utiliser une référence *locale* à votre fonction.

Question 8. Comparez l'efficacité des fonctions `monnaie_glouton`, `monnaie_rec` et `monnaie_memoizee`.

En plus du temps d'exécution, vous pourrez utiliser des compteurs pour compter le nombre d'appels à chacune des fonctions (pour des arguments identiques).

Quelles sont, à votre avis, leur complexités théoriques. (Essayez de justifier votre réponse.)

Question 9. Pour augmenter l'efficacité de la fonction, remplacer le module `Dict` par une instance du module `Map` de Caml en utilisant le foncteur `Map.Make` :

```
module M = struct type t=int*(int list) let compare = compare end
module Dict : TypeDict = Map.Make(M)
```

Expérimentez pour constater l'efficacité de la fonction `monnaie_memoizee` obtenue avec ce module.

---

* ou bien `monnaie_memoizee : int list -> int -> int` si vous n'avez pas fait la question 5.

Question 10. Pour finir, écrivez une fonction `test : int list -> int -> unit` qui fait la chose suivante :

- elle prend en argument en ensemble de pièces et une somme,
- elle calcule une solution en utilisant les fonctions `monnaie_glouton`, `monnaie_rec` et `monnaie_memoizee`,
- elle affiche un résumé de ce qui c'est passé. En particulier :
  - . elle affiche les 3 solutions trouvées,
  - . est-ce que l'algorithme `monnaie_glouton` donne une solution optimale,
  - . elle affiche le nombre d'appels récursifs pour chacune des fonctions,
  - . elle lève une exception si la solution proposée par `monnaie_glouton` est strictement meilleure qu'une des solutions proposée par `monnaie_rec` ou `monnaie_memoizee`.

Pour les affichages, vous pouvez utiliser les fonctions suivantes :

- `print_int : int -> unit`
- `print_string : string -> unit`
- `print_newline : string -> unit` (rajoute un saut de ligne à la fin)
- `Printf.printf` (très puissante, mais plus compliquée à utiliser)