TP 3— Programmation impérative, bibliothèque, tris

http://www.gallium.inria.fr/~scherer/teaching/colles/2012/tp3.pdf

Les questions en rapport avec les TP, Caml ou la programmation en général sont les bienvenues et peuvent être envoyées à gabriel.scherer@gmail.com. N'hésitez pas !

1 Programmation impérative

Nous allons maintenant récapituler rapidement les aspects "impératifs" de Caml Light: boucles (for, while) et champs modifiables (références, tableaux...). J'ai choisi de vous présenter tout ça seulement après la programmation récursive parce que c'est quelque chose que vous utilisez trop souvent[1], alors que les solutions sans boucles et sans références sont parfois plus simples et plus lisibles.

Je vous en parle maintenant parce qu'il arrive que ce soit la bonne solution, mais attention, ça ne veut pas dire qu'à partir de maintenant, vous devez vous en servir tout le temps !

1.1 Champs modifiables

En Caml, les variables ne sont pas *modifiables* : une fois qu'une variable a été déclarée par un let, sa valeur ne change pas (jusqu'à la déclaration suivante). Comme en mathématiques : vous n'écrivez jamais dans une copie "et maintenant n vaut n+1" !

Les informaticiens, contrairement aux mathématiciens, ont parfois besoin de *modifier* des valeurs (mais on peut s'en passer). Il existe donc plusieurs objets du langage Camlpermettant de faire cela.

Références : si 'a est un type Caml, 'a ref désigne le type des *références* contenant une valeur de type 'a. On peut voir une référence x comme un tiroir : on peut l'ouvrir pour regarder à l'intérieur (c'est !x), on voit un objet de type 'a, ou bien changer son contenu pour y mettre la valeur v (x := v).

Pour créer une nouvelle référence, on utilise la fonction ref en lui donnant une valeur de départ (la valeur que contiendra le tiroir avant que son contenu ne soit modifié). Attention à la subtilité : quand on utilise l'opérateur :=, on change le *contenu* de la référence, pas la référence elle-même (qui désigne toujours le même tiroir) !

```
let test =
  let a, b = ref 1, ref 2 in
  let c, d = a, ref a in
  a := !a + !b; d := c; !(!d) + !a;;
```

▷ **Question 1.** Quelle est la valeur de la variable test ? ◁

▷ **Question 2.** Que valent les variables test1 et test2 ? Que fait l'opérateur == ? ◁

```
let a, b = ref 1, ref 1;; let c = a;;
let test1 = (a = b, a == b);;
b := 2;;
let test2 = (a = b, a == c);;
```

Les cases des tableaux et des chaînes de caractères sont aussi des champs modifiables : tab.(i) et str.[i] permettent d'obtenir la valeur en i-ème position (en partant de 0) du tableau tab et de la chaîne str. Pour les modifier on n'utilise pas ":=" mais "<-" : tab.(i) <- v.

Les enregistrements (types produits utilisant la syntaxe { ... }) peuvent avoir certains champs modifiables, si c'est précisé à la déclaration par le mot-clé mutable :

```
type 'a reference = {mutable contents : 'a};;
```

On peut alors modifier le champ avec la syntaxe x.champ <- v.

▷ **Question 3.** On peut redéfinir les références en partant des champs modifiables. Définir get (pour !), set (pour :=) et la fonction new_ref (pour ref) en utilisant le type reference défini au dessus. ◁

1.2 Boucles

Sans surprise :

for <nom> = <expr1> to <expr2> do <expr3> done : faire parcourir les entiers de expr1 à expr3 à la variable nom en évaluant expr3 pour chaque valeur

while <expr1> do <expr2> done : évaluer expr2 tant que le booléen expr1 vaut true

▷ **Question 4.** Utiliser le crible d'eratosthène pour calculer la somme des nombres premiers entre 0 et 10^6. ◁

[1] Malheureux héritage de la programmation Maple, peut-être ?

1.3 Exceptions

Les exceptions sont une manière d'interrompre une partie d'un programme en cas d'erreur. Par exemple, si vous avez une grosse formule mathématique à calculer, et qu'en plein calcul vous vous apercevez que vous devez diviser 0, vous allez vous arrêter et vous plaindre que la formule n'est pas bien définie. `Caml Light`sait faire pareil.

Les expressions sont des objets de type `exn` qui ressemblent beaucoup aux types sommes définis dans le précédent TP : ce sont des constructeurs, qui peuvent comporter des arguments, et sont déclarés par le mot-clé `exception`.

```
#exception Erreur of string;;
let boum () = raise (Erreur "boum");;
try (boum (); "message") with
| Exit -> "sortie"
| Erreur message -> "erreur :  " ^message
| _ -> "exception inconnue"
```

Quand on a trouvé une erreur, on peut *lancer* une expression avec la fonction `raise` : elle prend une expression en paramètre, et interromp le calcul (en particulier, tout ce qui devait se passer ensuite dans le programme n'est pas exécuté).

Cela permet de faire des erreurs qui stoppent complètement le calcul. Parfois, on voudrait plutôt détecter l'erreur et utiliser une solution adaptée pour continuer le programme (par exemple si l'erreur est "plus de papier dans l'imprimante", il suffit de demander à l'utilisateur de rajouter du papier avant de continuer, au lieu d'annuler complètement l'impression en cours). On peut *rattrapper* une exception avec la construction `try <expr> with <filtrage>`. Cela se présente un peu comme un `match .. with`, mais le comportement est différent :

- si l'évaluation de `<expr>` ne provoque aucune exception, on renvoie sa valeur
- sinon, on effectue le filtrage sur la valeur de l'exception envoyée

Plusieurs exceptions sont prédéfinies en `Caml Light`(vous les trouverez dans le manuel, module `exc`). Vous rencontrerez (et/ou utiliserez) sans doute entre autres `Exit`, qui permet simplement de sortir d'un calcul, et `Failure of string` qui permet d'indiquer la cause de l'erreur. On peut aussi utiliser la fonction `failwith` qui lance `Failure`, et est définie ainsi : `let failwith str = raise (Failure str)`

```
let existe element tableau =
  let i, trouve = ref 0, ref false in
  while not !trouve && !i < vect_length tableau do
    trouve := tableau.(!i) = element;
    incr i
  done;
  !trouve
```

▷ **Question 5.** Réécrire la fonction `existe` pour utiliser une boucle `for`, sans parcourir plus d'éléments du tableau. ◁

▷ **Question 6.** Quel est le type de `raise` ? Pourquoi ? ◁

1.4 Dangers

On veut créer un tableau à deux dimensions, sans utiliser[2] la fonction déjà toute faite `make_matrix`.

```
let nouvelle_matrice n p x =
  make_vect n (make_vect p x);;
let mat = nouvelle_matrice 3 3 0;;
mat.(0).(1) <- 2; mat;;
```

▷ **Question 7.** Quel est le problème avec `mat` ?
Coder (correctement) `nouvelle_matrice` avec une boucle.
Coder `nouvelle_matrice` avec `init_vect`.
Coder `nouveau_pave` qui crée un tableau en trois dimensions. ◁

2 Boite à outils

Vous avez globalement vu l'ensemble des concepts `Caml Light`dont vous aurez besoin. Voici maintenant quelques *bibliothèques*, des ensembles de fonctions déjà existantes (vous en avez déjà rencontrées plusieurs) qu'il est parfois utile de pouvoir utiliser sans les recoder.

Pour s'informer, la meilleure solution est d'aller consulter la section du manuel : http://caml.inria.fr/pub/docs/man
caml-light/

Dans cette partie, vous aurez besoin de consulter la documentation. Si vous n'avez pas d'accès internet, c'est le moment de se plaindre.

[2] ça forge le caractère

Les fonctions de ces bibliothèques sont organisés dans des *modules* thématiques. Par exemple le module `list` contient toutes les fonctions pour manipuler des listes.

On distingue deux types de modules :

- les modules "core" : les fonctions des modules du core sont accessibles directement par leur nom, par exemple `map` dans le module `list`
- les modules "standard" : les fonctions des modules standards doivent être préfixées par le nom du module : pour appeler la fonction `init` du module `random`, on utilise `random__init`. Notez le *double* underscore.

Il est possible de se passer du nom du module en préfixe en utilisant une directive #open. Dans la suite du code il suffit alors de mettre le nom des fonctions du module, comme pour les modules cores.

```
random__init ();;
#open "random";;
init ();;
```

2.1 STACK, QUEUE

`stack` contient des fonctions pour manipuler des *piles* (premier entré dernier sorti), et `queue` des *files* (premier entré premier sorti). Ces structures de données se manipulent de façon impérative.

▷ **Question 8.** Écrire des fonctions de conversion de pile en liste (et réciproquement) et de file en liste (idem). ◁

2.2 SET, MAP

`set` et `map` sont des structures de données plus sophistiquées, qui permettent de stocker des éléments et d'y accéder rapidement (en un temps logarithmique). `set` représente un ensemble (on peut mettre des éléments et se demander des éléments sont dedans) et `map` une table d'association : comme un tableau, mais au lieu d'indexer par un entier on peut indexer par n'importe quel type dont on sait comparer les éléments.

Pour faire des tables d'association, on peut aussi utiliser le module de tables de hachage `hashtbl`. Il est plus adapté aux contextes impératifs, et `map` aux contextes fonctionnels/récursifs (sans modification de valeurs).

▷ **Question 9.** Écrire une fonction qui ajoute les entiers de 0 à 10000 dans une liste, puis teste pour chaque entier de 0 à 10000 s'il appartient à la liste.

Faire de même avec un `set`, et comparer les temps d'exécution. ◁

▷ **Question 10.** La complexité exponentielle de la version naïve de fibonnaci vient du fait que la fonction est rappellée un grand nombre de fois avec les mêmes arguments. On peut l'optimiser naïvement de la manière suivante : on crée une fonction fibonacci qui se "souvient" des réponses qu'elle a déjà donné : si on lui demande la valeur en un nombre pour la première fois, elle calcule le résultat et le stocke dans un "cache" (une `map`), et ensuite elle redonne directement le résultat calculé. Implémenter une telle fonction fibonnaci optimisée.

Implementer une fonction `memoize : ('a -> 'b) -> ('a -> 'b)` qui est capable de "mettre en cache" n'importe quelle fonction (on utilisera pour créer la `map` la fonction de comparaison générique `compare`). ◁

2.3 RANDOM

▷ **Question 11.** Écrire une fonction qui permute aléatoirement tous les éléments d'un tableau. ◁

2.4 PRINTF

Printf est un module d'affichage polyvalent. Il est aussi pratique que sa documentation est indéchiffrable.

```
#open "printf";;
printf "%d\n" 5;;
printf "J'ai %d %ss !\n" 5 "pomme";;
```

▷ **Question 12.** Aller lire, et essayer de comprendre une partie de, la documentation du module `printf`. Que fait la fonction `sprintf` ? ◁

3 Questions difficiles

▷ **Question 13.** Écrire une fonction `finie` qui renvoie vrai si et seulement si la liste passée en argument est de longueur finie (et qui termine dans tous les cas, bien sûr). ◁

On essaiera dans cette partie de définir des fonctions récursives sans utiliser le mot-clé `rec`. C'est inutile en `Caml`, mais cela repose sur des idées intéressantes, et ce n'est pas facile.

3.1 Dé-récursifier

On travaillera sur l'emblématique fonction `fac` : `let rec fac n = if n = 0 then 1 else n * fac (n - 1)`
On définit la fonction `fix` suivante : `let rec fix f = fun x -> f (fix f) x`

▷ **Question 14.** Écrire une fonction non récursive `fac_derec` telle qu'on puisse définir `fac` ainsi : `let fac = fix fac_derec;;` ◁

`fac_derec` représente une version "dérécursifiée" (sans appel récursif) de `fac`. On peut systématiquement dérécursifier les fonctions récursive, et retrouver le comportement initial en utilisant `fix`.

On a ici séparé deux choses : le corps de la fonction (qui effectue le vrai calcul) et l'application de la récursivité (qui est reléguée à `fix`, qui est elle-même définie en utilisant `rec`). On cherche maintenant à écrire d'autres versions de `fix`, qui font la même chose sans utiliser `rec`.

3.2 Récursion par effet de bord

▷ **Question 15.** Écrire une version de `fix` utilisant des références.
Voici un squelette de code à compléter :

```
let fix f = let res = ref (fun _ -> hd []) in ...  ; !res
```
◁

3.3 Récursion par type récursif

▷ **Question 16.** On définit les fonctions suivantes :

```
let auto x = x x
let fix f = auto (fun x -> f (auto x))
```

Montrer en utilisant un papier et un stylo, et en faisant les réductions à la main, que `fix f = f (fix f)`.
Pourquoi ça ne marche pas en `Caml Light`? ◁

▷ **Question 17.** L'astuce est d'utiliser un *type récursif*.
Utiliser le type suivant pour définir une version "qui marche" de `auto` :

```
type 'b auto = Auto of ('b auto -> 'b)
```

Pourquoi a-t-on choisi cette définition du type `'b auto` ? ◁

▷ **Question 18.** Définir correctement (en utilisant `auto`) le `fix` de la question 16. Tester avec `fac`.
Utiliser plutôt `let fix f = auto (fun x -> f (fun y -> auto x y))`. ◁