L3 ENS Cachan - Prog 1

Soutient OCaml

d'après des TPs de David Baelde

1 Des bases

On commence par un petit retour aux bases : les fonctions et les types d'OCaml. Pour la section 1.1, vous pouvez utiliser le *top level* d'OCaml (vous y avez accès en tapant `ledit ocaml`). Pour la suite des exos, écrivez le code dans un fichier et compilez comme on a appris.

1.1 Des fonctions

Nous allons bientôt définir nos premiers programmes sous la forme de fonctions. Voyons rapidement différentes façons de définir ou utiliser une fonction. En Caml il n'y a pas de différence entre `(f(x))` et `(f x)` mais il y en a une entre `(f x y)` et `(f (x,y))`. Comparons les deux définitions suivantes, toutes deux valides :

```
let f1 x y = x+y ;;
let f2 (x,y) = x+y ;;
f1 2 3 ;;
f2 (2,3) ;;
f1 (2,3) ;; <-- invalide !
```

▶ Mais ce n'est pas tout. Évaluez `let f3 = f1 1`, comprenez ce que fait `f3`.

Pour mieux comprendre, il faut savoir que `f1` peut aussi s'écrire `fun x -> fun y -> x+y`, ce qui met bien en évidence la possibilité d'*application partielle*. Cette version de la fonction est dite *curryfiée*, et c'est la forme qu'on préferera en général. Dans le même ordre d'idée on notera l'associativité à droite de la flèche : `a -> b -> c = a -> (b -> c)`.

1.1.1 La factorielle

Pourquoi faut il déclarer les fonctions récursives ? En Caml, on procède par définition plutôt que par instantiation. Quand on écrit `let a = ..` il ne s'agit pas de modifier `a` mais de le (re-)définir. On écrit ainsi souvent `let a = .. in let a = ..a.. in ..` pour expliciter des étapes du calcul de `a`. Il est donc pratique que la référence à `a` dans la redéfinition de `a` soit relative à la première définition. C'est pourquoi la récursivité doit être spécifiée explicitement au moyen de `let rec a = ..`, quand on veut que les références à `a` parlent de la définition en cours.

▶ Définissez la fonction factorielle, et testez-la sur quelques valeurs. Remarquez que les valeurs trop grandes deviennent fantaisistes, négatives ou nulles. Vous devez avoir une vague idée de pourquoi.

Pour les curieux rapides : essayez de calculer 100000!. Une implémentation naïve échoue avec le message `Stack overflow during evaluation (looping recursion?)`.
On peut éviter ces erreurs en programmant ses fonctions récursives de façon à ce qu'elles soient *récursives terminales*. Dans ce cas précis ça ne nous avancerait pas, car on a compris que $100000! = 0$, mais cette technique sert aussi à rendre les calculs plus rapides. Le corrigé contient une solution récursive terminale.

1.1.2 En finir avec les types

Vous n'en aurez probablement pas vraiment fini de vous débattre avec les erreurs de type, mais voici en tout cas le premier et dernier exercice dédié au sujet.

▶ Pour chacun des types suivants, définissez une valeur ayant ce type :
- `'a -> 'a`
- `'a -> 'b -> 'a`
- `'a -> 'b -> 'a * 'b`
- `'a * 'b -> 'a`
- `('a -> 'b -> 'c) -> 'a * 'b -> 'c`
- `('a * 'b -> 'c) -> 'a -> 'b -> 'c`
- `('a -> 'b -> 'c) -> 'b -> 'a -> 'c`

1.2 Variants

Les types variants sont un outil très pratique pour définir des types de donnée. L'utilisation des filtrages permet de les traiter de façon intuitive, et le compilateur Caml vous aide à ne pas oublier de traiter un cas. Peu de langages fournissent un outil aussi expressif.

▶ Définir à l'aide de types variants les entiers de Peano, puis les listes. Si vous ne vous sentez pas à l'aise, écrivez la traduction des entiers de Peano vers `int`, ou la fonction `map : ('a -> 'b) -> 'a list -> 'b list` telle que :

`map f [ a_1 ; .. ; a_n ] = [ f a_1 ; .. ; f a_n ]`.

Sinon vous pouvez passer tout de suite à la suite, plus originale.

1.2.1 Entiers naturels en binaire

On représente les entiers naturels strictement positifs codés en binaire par le type variant suivant : `type entier = H | O of entier | I of entier`

Le nombre 10010 sera représenté par `O(I(O(O(H))))`.

▶ Définissez les fonctions de conversion entre `entier` et `int` :

```
int_of_entier : entier -> int
entier_of_int : int -> entier
```

▶ Définissez l'addition sur les `entier`. Vous n'utiliserez pas de conversion vers `int` mais pourrez définir une fonction auxiliaire de type `entier -> entier -> bool -> entier` dont l'argument booléen indique s'il faut propager une retenue.

Vérifiez quelques additions en utilisant les conversions :

```
let _ =
  for i = 1 to 16 do
    for j = 1 to 16 do
      let k = int_of_entier
                (somme (entier_of_int i) (entier_of_int j)) in
        if k <> i+j then
          Printf.printf "Erreur: %d+%d = %d ??!\n" i j k
    done
  done ;
  Printf.printf "Fin du test.\n"
;;
```

1.3 Listes associatives triées

Avant de construire et manipuler des arbres, nous allons créer un outil préliminaire : des listes associatives triées. Une liste associative est une liste de couples vue comme un ensemble d'associations $k \mapsto v$: `[ (k_1, v_1) ; .. ; (k_n, v_n) ]`.

La librairie standard de Caml fournit des fonctions pour manipuler de telles listes :

```
# List.assoc 3 [ 1,"un" ; 2,"deux" ; 12,"douze" ] ;;
- : string = "deux"
```

En guise d'exercice nous allons ré-écrire les fonctions de manipulations de listes associatives de façon plus efficace, puisque nous maintiendrons des listes triées. De plus nous assurerons qu'il y a au plus une association par clé.

Pour vous simplifier la tâche, nous nous restreindrons au cas où la fonction de comparaison pour les tris est `<`. Mais les clés et les valeurs associées sont de type quelconque, j'insisterai donc sur le polymorphisme dans les types de fonctions demandées.

Nous utiliserons le type `option` pour gérer le cas où aucune association n'est trouvée, au lieu des exceptions que vous n'avez pas encore abordées. Ce type est déja défini dans la librairie standard de Caml, comme suit :

```
type 'a option = None | Some of 'a
```

Le type option sert quand on n'est pas sûr de pouvoir renvoyer une valeur du bon type. On renvoie `Some v` quand on le peut, et `None` sinon.

> **Les types paramétrés.** Vous découvrez peut-être les types paramétrés, le *polymorphisme de type*. Sur l'exemple précédent cela signifie simplement qu'on peut utiliser le constructeur `Some` sur n'importe quel paramètre de type `t`, le résultat sera alors de type `t option`.
>
> ```
> # Some 12 ;;
> - : int option = Some 12
> # Some "douze" ;;
> - : string option = Some "douze"
> # None ;;
> - : 'a option = None
> ```

▶ Ecrivez une fonction `assoc` de type `('a*'b) list -> 'a -> 'b option`, telle que `assoc l k` renvoie le premier `v` tel que `k,v` est dans `l`. Vous supposerez que la liste est triée par ordre croissant (pour la fonction de comparaison `<`) et veillerez à en tirer parti.

▶ Ecrivez une fonction d'insertion `add_assoc` de type `('a*'b) list -> 'a -> 'b -> ('a*'b) list` telle que pour toute liste d'association triée `l`, `add_assoc l k v` renvoie la liste d'association triée contenant les mêmes associations que `l`, sauf pour `k` auquel sera associé `v`. L'association éventuelle de `k` dans `l` sera effacée.

▶ Ecrivez une fonction qui à une liste associative non triée associe la liste associative triée, en utilisant l'insertion triée. Quand plusieurs associations apparaissent dans la liste non triée, laquelle est préservée ? Savez vous évaluer la complexité de cet algorithme de tri ?

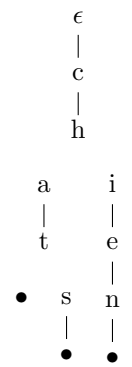
2 Dico

L'objectif ici est de développer une représentation pratique d'un ensemble de mot, sous forme d'un arbre, en mettant en oeuvre l'insertion et la recherche d'un mot. A la fin, nous calculerons des préfixes univoques minimaux dans un dictionnaire.

2.1 Arbres dictionnaires

Donnons nous un *alphabet* A, *a priori* un ensemble quelconque mais ici l'alphabet romain, ou les valeurs de type `char` en Caml. Un *mot* sur l'alphabet A est une suite d'éléments de A. Le mot vide est noté ϵ. Un *langage* est un ensemble de mots. On note cw le mot $cc_1 \cdots c_n$ si w est le mot $c_1 \cdots c_n$. Pour un langage A on notera par extension $cA = \{cw | w \in A\}$.

Un dictionnaire est une représentation d'un ensemble de mots permettant une recherche facile. Nous allons utiliser pour cela une représentation arborescente. Par exemple, l'arbre suivant représente l'ensemble $\{chien, chat, chats\}$.



► Montrez que tout langage peut s'écrire sous la forme $E \cup \bigcup_{1 \leq i \leq n} c_i A_i$ où E est vide ou restreint à $\{\epsilon\}$, les A_i sont des langages et les c_i sont des caractères distincts.

Par induction sur la taille d'un langage L en utilisant la question précédente, nous savons maintenant que tout langage peut se représenter par une valeur du type suivant en Caml. La liste d'associations en paramètre représente les c_i, A_i. Le constructeur `Plein` dénotera la présence de ϵ dans l'ensemble, `Vide` sera utilisé pour un langage ne contenant pas le mot vide.

```
type dico = Plein of (char*dico) list | Vide of (char*dico) list

(* On retrouve ainsi l'exemple précédent *)
let exemple =
  Vide [('c', Vide [('h', Vide
         [('a', Vide [('t', Plein [('s', Plein [])])]);
          ('i', Vide [('e', Vide [('n', Plein [])])])])])]
```

2.2 Insertion et recherche

Nous programmons maintenant les fonctions élémentaires de manipulation de dictionnaires. Vous les testerez sur quelques exemples couvrant plusieurs cas.

► Programmez la fonction `appartient` de type `dico -> char list -> bool` qui indique si un mot est présent dans un dictionnaire.

▶ Définissez maintenant la fonction `insertion` de type `dico -> char list -> dico` telle que `insertion dico mot` renvoie le nouveau dictionnaire correspondant à `dico` auquel on a ajouté `mot`.

2.3 Préfixes univoques minimaux

Pour finir nous allons calculer l'ensemble des *préfixes univoques minimaux* d'un langage. Un *préfixe* de $c_1 \cdots c_n$ est un mot de la forme $c_1 \cdots c_i$ pour $0 \leq i \leq n$. Le préfixe est dit strict si $i < n$. Le préfixe u est dit *univoque* dans un langage L s'il est préfixe d'un unique mot du langage L. Enfin, il est *minimal* s'il n'a pas de préfixe strict qui soit encore univoque.

En français il s'agit des plus courtes abbréviations d'un mot qui ne soient pas ambigues. Pour l'exemple de langage cité plus haut, l'ensemble des préfixes univoques minimaux est : $\{chi, chats\}$

Nous allons voir que la structure de dictionnaire rend facile le calcul de l'ensemble des préfixes univoques minimaux d'un langage, noté $pum(L)$.

▶ Montrez que :
— $pum(\{\epsilon\}) = \{\epsilon\}$
— $\forall L \forall c, pum(L) = \{\epsilon\} \Rightarrow pum(cL) = \{\epsilon\}$

▶ Pour un langage $L = E \cup \bigcup_{1 \leq i \leq n} c_i L_i$ avec $n > 1$, exprimez $pum(L)$ en fonction des $pum(L_i)$.

▶ Exprimez $pum(L)$ en fonction de $pum(L_1)$ pour
— $L = c_1 L_1$ quand $pum(L_1) \neq \{\epsilon\}$
— $L = \{\epsilon\} \cup c_1 L_1$

▶ Vous pouvez maintenant définir la fonction `univoques` de type `dico -> char list list` qui renvoie la liste des préfixes univoques minimaux d'un dictionnaire.