TP1 Caml : prise en main

Sujet à finir pour dimanche 23 septembre 2014 à 20h00.
À rendre par mail à `aurelie.lagoutte@ens-lyon.fr`

Caml est un langage de programmation développé à l'INRIA. Il se prête principalement au style de programmation fonctionnel, mais également si besoin aux styles impératif et même objet. C'est un langage fortement typé, où les incompatibilités de types sont détectées à la compilation, et où les conversions implicites de types sont impossibles.
De nombreuses informations sont disponibles en ligne, en particulier à partir du site officiel `http://caml.inria.fr/` (vous trouverez notamment un résumé de syntaxe `Caml` à `http://caml.inria.fr/pub/old_caml_site/FAQ/qrg-fra.html`).

Environnement

L'interpréteur interactif (*top-level* en anglais) `ocaml` peut s'utiliser de deux façons : dans un terminal (`xterm` par exemple) et dans l'éditeur de texte Emacs.

Utiliser `ocaml` dans Emacs. L'éditeur de texte Emacs peut interagir avec `ocaml`. On peut donc à la fois utiliser `ocaml` et sauvegarder son programme. Ouvrez Emacs, ouvrez un nouveau fichier auquel vous donnerez l'extension `.ml`. Tapez la ligne de test `1+1;;` dans votre nouveau fichier. Comme Emacs reconnaît l'extension `.ml`, il lance automatiquement le mode Tuareg et son menu. Maintenant, il nous faut une fenêtre pour lancer Caml. Couper la fenêtre existante en deux (si ce n'est pas déjà le cas) en utilisant le raccourci CTRL-X 2. Evaluer maintenant votre ligne de test avec le raccourci CTRL-X CTRL-E, Emacs propose alors de lancer ocaml : on valide, et on admire le résultat de l'évaluation Caml. On peut maintenant écrire dans notre fichier (en haut) et, à chaque fois que l'on souhaite, évaluer la ligne courante grâce au raccourci CTRL-X CTRL-E. On peut interrompre le top-level lors d'un calcul trop long grâce au raccourci CTRL-X TAB, et on peut le fermer avec CTRL-C CTRL-K. Le raccourci pour sauver son fichier .ml est CTRL-X CTRL-S (à entrer quand on se trouve dans la partie du haut). À utiliser sans modération. Le raccourci clavier pour fermer Emacs est CTRL-X CTRL-C.

Utiliser `ocaml` dans un terminal (pour les geeks). Il suffit d'utiliser la commande `ocaml` dans le terminal, pour démarrer l'interpréteur interactif en ligne de commande. Pour pouvoir se déplacer dans la ligne de commande et revenir dans l'historique, on lancera `ledit ocaml`, la commande `ledit` permettant d'éditer des lignes en ligne de commande. Une fois que `ocaml` est lancé, une invite de commande (*prompt*) # et un curseur s'affichent. On entre le programme et on valide avec la touche Entrée. On arrête un calcul avec CTRL-C et on quitte avec CTRL-D.

Dans tous les cas Il est important de bien indenter son code (c'est-à-dire laisser des espaces en début de ligne de façon à aligner intelligemment certaines lignes de code, par exemple pour le filtrage, pour le début et la fin d'une boucle, etc...). Ceci est vrai dans tous les langages. Pour vous aider à faire cela correctement en Caml, Emacs propose de le faire automatiquement pour la ligne en cours lorsque vous tapez sur TAB. Pensez-y, c'est pratique ! Cela permet parfois de détecter des erreurs avant même la compilation, si l'indentation automatique ne correspond pas à ce que vous espériez (oubli d'un point-virgule, ou d'une parenthèse, etc...).

1 Premiers pas

Entrez les expressions suivantes dans le top-level, et essayez à chaque fois d'interpréter et d'expliquer la réponse d'`ocaml`. Comme évoqué précédemment, Caml est un langage fortement typé, c'est-à-dire qu'il est capable de donner un type à chaque variable ou fonction, et vous l'affiche comme réponse après une évaluation. Il est très important de comprendre les réponses de Caml pour vérifier que le programme que l'on a écrit correspond à ce que l'on voulait dire, et c'est également une aide utile pour le débuggage.

Les types de base en Caml sont :

`Type Caml`	Signification
`int`	Entier avec signe 31 bits (environ +/- 1 milliard) ou bien 63 bits, selon le processeur
`float`	Nombre à virgule flottante
`bool`	Booléen, noté `true` ou `false`
`char`	Un caractère à 8 bits
`string`	Une chaîne de caractères à 8 bits
`unit`	Valeur "rien" notée `()`
`'a list`	Liste d'éléments de type `'a`, où `'a` remplace n'importe quel type constructible

1.1 Évaluation d'expressions

```
51;;
16 - 64;;
32 *
(51+1);;
8 / 3;;
3.14;;
2 * 3.14;;
2. *. 3.14;;
"Hello World!";;
("mdr",3);;
("mdr,3");;
"Truc\n";;
print_string "Truc\n";;
true;;
true && false;;
true || false;;
2+2 = 4;;
if 3*3 > 8 then "Truc" else "Bidule";;
if false || not true then "un" else 2;;
2 * (* ceci est un commentaire *) 3;;
(*les utiliser sans moderation !!!*) 1+1;;
```

1.2 Déclaration de variables, globale et locale

```
let a = 4;;
a;;
let b = 13 * a;;
b;;
let a = a * a;;
a;;
```

```
b;;
let c = 2 in a * b * c;;
c;;
let a = 21 in a * 2;;
a;;
let (a,b) = (3,4) in a * b;;
```

1.3 Fonctions et fonctions récursives

```
let carre x = x * x;;
carre 4;;

let mult x y = x * y;;
mult 3 4;;

let triple = mult 3;;
triple 14;;

let mult_bis (x,y) = x*y;;
mult_bis (3,4);;
let triple_bis y = mult_bis (3,y);;

let rec plus x y = if y = 0 then x else plus (x+1) (y-1);;
plus 3 0;;
plus 3 4;;
plus 3 (-1);;  (* oups ! *)

let rec f x = f x;;
f 0;; (* re-oups ! *)

(fun x -> x*x*x) 4;;

let apply_rev f x y = f y x;;
let g x y = (x,y);;
apply_rev g 64 16;;
apply_rev (fun x y -> x / y) 3 18;;

let rec f x = (x x);;
```

1.4 Listes

Une liste est une collection ordonnée d'éléments, dont on peut décrire récursivement la construction : une liste `l` est soit vide, dans ce cas elle est notée `[]`, soit formée d'un premier élément `t` (la *tête* de la liste), et d'une liste `q` contenant le reste de `l` (la *queue* de la liste). Elle est alors notée `t::q`. Caml nous permet de faire du *filtrage* sur la forme d'une variable de manière très pratique (sans utiliser de `if...`), comme illustré sur l'exemple suivant.

```
let est_vide l = match l with
 |[]->true
 |_-> false;;
```

On remarquera que "_" signifie "n'importe quoi d'autre". De plus, lorsque le filtrage porte sur le dernier argument de la fonction, il est plus élégant d'utiliser le filtrage directement :

```
let est_vide = function
 |[]->true
 |_-> false;;

let rec produit = function
|[] -> 1
|x::xs -> x * (produit xs);;

produit [2;8;4;2;13];;

let un_seul_element=function
 |[a]->true
 |_->false;;

 un_seul_element [3];;
 un_seul_element [];;
 un_seul_element [1;2;3];;

let rec garde_positifs = function
 |[]->[]
 |t::q when t>=0 -> t::(garde_positifs q)
 |t::q->garde_positifs q;;

 garde_positifs [2;4;-3;6;-11];;
```

Le dernier exemple nous montre l'importance de l'ordre dans lequel on écrit les différents cas du filtrage.

2 Exercices

N'oubliez pas que chaque fonction écrite doit correspondre au type spécifié dans l'énoncé, et doit être accompagnée d'un ou plusieurs exemples l'illustrant. En particulier lors d'un filtrage, chaque cas doit être testé. Tout raisonnement non-trivial sera expliqué en commentaires.

2.1 Fonctions

Exercice 1 (Factorielle) *Écrivez récursivement la fonction factorielle* `fact:int -> int` *telle que* `fact n` *renvoie* $n!$ *Attention au cas où* n *est négatif (n pourra dans ce cas utiliser* `failwith "Nombre negatif"`*).*

Exercice 2 (Codage binaire) *Écrivez une fonction* `bin:int -> int` *qui renvoie le codage binaire d'un entier* n*.*

Exercice 3 (Suite de Fibonacci)

1. *Écrivez récursivement la fonction* `fibo:int -> int` *telle que* `fibo n` *renvoie le terme* u_n *de la suite de Fibonacci definie par* $u_0 = 0$, $u_1 = 1$ *et* $u_n = u_{n-1} + u_{n-2}$ *pour* $n \geq 2$.
2. *Comptez le nombre d'appels récursifs à cette fonction lors de l'évaluation de* `fibo n`.
3. *Écrivez une fonction récursive* `fibo_aux:int -> int * int` *telle que* `fibo_aux n` *renvoie le couple* (u_{n-1}, u_n) *en seulement* n *appels récursifs.*
4. *Écrivez une fonction* `fibo2:int -> int` *faisant appel à* `fibo_aux` *pour calculer les termes de la suite de Fibonacci.*
   Remarque : dans la vraie vie, on utilisera la construction suivante :

```
let fibo2 n =
let fibo_aux n =
...
in
...;;
```

   Ainsi, `fibo_aux` *n'existe pas en-dehors de* `fibo2`.
5. *Comparer le temps de calcul de* `fibo 40` *et* `fibo2 40`.

Exercice 4 (Types de fonctions) *Écrivez une fonction ayant pour type* `'a -> 'a -> 'a`, *puis une autre ayant pour type* `('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c`.

2.2 Manipulation de listes

Exercice 5 (Fonctions élémentaires)

1. *À l'aide du filtrage, écrivez la fonction* `hd:'a list -> 'a` *qui renvoie le premier élément de la liste. (Utilisez* `failwith` *si la liste est vide.) De même, écrivez la fonction* `tl:'a list -> 'a list` *qui renvoie la queue de la liste (la liste privée de son premier élément).*
2. *Écrivez la fonction* `length:'a list -> int` *qui calcule la longueur de la liste.*
3. *Écrivez la fonction* `nth:'a list -> int -> 'a` *qui renvoie le n-ème élément de la liste. N'oubliez pas que le filtrage sur plusieurs arguments, par exemple* `n` *et* `l`, *est possible grâce à l'opérateur* `match (n,l) with ....`
4. *Écrivez la fonction* `rev:'a list -> 'a list` *qui renverse la liste. (Astuce : écrire une fonction* `rev_aux:'a list -> 'a list -> 'a list`, *telle qu'un appel à* `rev_aux x::l1 l2` *fasse un appel récursif à* `rev_aux l1 x::l2`. *La liste* `l2` *joue ici un rôle d'accumulateur.*
5. *Écrivez la fonction* `append:'a list -> 'a list -> 'a list` *qui concatène les deux listes.*
6. *Écrivez la fonction* `map:('a -> 'b) -> 'a list -> 'b list` *telle que* `map f [a1;...; an]` *renvoie la liste* `[f a1;...; f an]`.

Exercice 6 (Pour tout et il existe)

1. *Écrivez la fonction* `fold:('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` *telle que* `fold f a [b1;...; bn]` *renvoie* `f (...(f (f a b1) b2)...) bn`.
   Exemple : `fold (fun s x -> 10*s+x) 0 [1; 2; 3; 4]` *renvoie* `1234`.
2. *Écrivez la fonction* `for_all:('a -> bool) -> 'a list -> bool` *telle que* `for_all p [a1; ...;an]` *renvoie* `true` *si et seulement si tous les éléments la liste satisfont le prédicat* `p`, *soit* `(p a1) && (p a2) &&...&& (p an)`. *Vous ferez deux versions : une version directe et une version utilisant* `fold`.

3. *Même question pour la fonction* `exists : ('a -> bool) -> 'a list -> bool` *qui renvoie* `true` *si et seulement s'il existe un élément de la liste qui satisfait* `p`*. Vous ferez deux versions : une version directe et une version utilisant* `fold`*.*

Exercice 7 (Tri fusion)

1. *Écrivez une fonction de tri d'une liste d'entiers qui implante l'algorithme de tri-fusion (fonction* `tri_fusion: 'a list -> 'a list`*). Cet algorithme repose sur une approche diviser pour régner : si une liste contient deux éléments ou plus, il suffit de la partager en deux listes de tailles (environ) la moitié de la liste de départ (fonction* `split: 'a list->'a list * 'a list`*), qui seront triées récursivement, puis fusionnées (fonction* `fusionne: 'a list -> 'a list-> 'a list`*). On se souviendra de la construction* `let ... in` *pour définir une variable locale.*
2. *Que nous apprend le typage de la fonction* `fusionne` *sur l'utilisation de l'opérateur de comparaison* `<` *en Caml ?*

Revenons aux listes traditionnelles. Lorsque deux cas du filtrage (aussi appelé *pattern-maching*) doivent renvoyer la même réponse, on peut les accoler sur la même ligne comme en témoigne l'exemple suivant :

```
let au_moins_deux = function
 | [] | [_] -> false
 | t1::t2::q -> true;;
```

Remarquez en revanche que l'exemple suivant renvoie une erreur. Comprenez-vous pourquoi ?

```
let au_moins_deux = function
 | [] | [a] -> false
 | t1::t2::q -> true;;
```

Exercice 8
1. *Écrivez une fonction* `extremum: 'a list -> 'a * 'a` *qui renvoie le couple (min, max) des élements de la liste.*
2. *Écrivez une fonction* `deuxieme: 'a list -> 'a` *qui renvoie le deuxième plus grand élément de la liste. On pourra s'aider d'une fonction auxiliaire qui renvoie une information plus forte.*

Exercice 9
1. *Écrivez une fonction* `map: ('a -> 'b) -> 'a list -> 'b list` *telle que* `map f [a1; ...; an]` *renvoie* `[f(a1); ...; f(an)]`*.*
2. *Texte à trous :*

```
map ............ [4; 7; -5 ; 2; 0];;
- : int list= [6; 9; -3; 4; 2]

map (fun n-> fibo n) .............. ;;
- : int list = [1; 2; 3; 5; 8]
```

2.3 Les arbres

On peut effectuer en `Caml` des déclarations de types, c'est-à-dire créer de nouveaux types de valeur. Les arbres binaires étiquetés par des entiers peuvent par exemple être représentés en `Caml` à l'aide de la déclaration de type suivante :

```
type arbre =
|Vide
| Noeud of int * arbre * arbre ;;
```

`Vide` et `Noeud` seront appelés les constructeurs du type `arbre`. Ils doivent commencer par des majuscules. Remarquons que dans ce modèle, une feuille est un `Noeud` dont les deux fils sont `Vide`. Une fois ce type déclaré, on peut construire des valeurs de type `arbre` :

```
Vide ;;
Noeud (42, Vide, Vide) ;;
```

Exercice 10 *Écrire en* `Caml` *les fonctions suivantes :*
- `taille : arbre -> int` *qui renvoie le nombre de noeuds internes (en comptant les feuilles)*
- `nb_feuilles: arbre-> int` *qui renvoie le nombre de feuilles*
- `hauteur : arbre -> int` *qui renvoie la hauteur de l'arbre, c'est-à-dire la longueur de sa plus grande branche*
- `detect : arbre -> int -> bool` *qui renvoie true si et seulement si l'un des noeuds de l'arbre a est étiqueté par n.*
- `complet : arbre -> bool` *déterminant si un arbre est complet, c'est-à-dire si toutes ses feuilles sont à la même profondeur.*

Si l'on ne sait pas à l'avance quel sera le type des étiquettes de notre arbre, on peut utiliser un type *polymorphe* :

```
type 'a arbre=
  | Vide
  | Noeud of 'a* 'a arbre * 'a arbre;;

let a1=Noeud(true,Vide, Noeud(false,Vide,Vide));;

let a2=Noeud("oui",Vide, Noeud("non",Vide,Vide));;

let a3=Noeud(3,Vide, Noeud(false,Vide,Vide));; (*pourquoi ? *)
```

Exercice 11 *Ecrivez un type polymorphe* `'a pile` *qui modélise une pile (on distinguera les constructeurs selon que la pile est vide ou non). Ecrivez aussi les fonctions suivantes :*
- `pile_vide: 'a pile->bool`
- `depiler: 'a pile -> 'a * 'a pile`
- `empiler: 'a -> 'a pile -> 'a pile`