Agrégation de Mathématiques option Informatique
TP1

Loïg Jezequel
loig.jezequel@irisa.fr

Mardi 13 Septembre 2011

Ce TP – largement inspiré des premiers chapitres du livre *Le Langage Caml* de Pierre Weis et Xavier Leroy – présente les bases du langage Caml : définitions, déclarations de fonctions, filtrage, listes, déclarations de types...

Question 1. Dans un premier temps nous allons utiliser un interpréteur Caml depuis un terminal. Pour cela ouvrez un terminal puis utilisez la commande `ocaml`. Essayez de faire évaluer quelques expressions (par exemple $1+1$ ou $3-2$). Chaque expression à évaluer doit être suivie de `;;` pour signifier à l'interpréteur où elle termine.

1 Définitions globales et locales

En Caml on donne un nom à une valeur en utilisant le mot clé `let`, par exemple :

```
let x = 3;;
let y = 4 + 5;;
```

Il est alors possible d'accéder a ces valeurs par leur nom, par exemple :

```
x + y;;
```

Ces valeurs ont été définies globalement. Il est aussi possible de définir des valeurs localement grace au mot clé `in`, par exemple :

```
let tmp = 3 in tmp + 1;;
```

Il n'est alors pas possible d'accéder a ces valeurs par leur nom en dehors de l'expression qui suit le `in`. On peut aussi réaliser des définitions simultanées par le mot clé `and`, par exemple :

```
let u = 1 and v = 2 in u + v;;
```

C'est toujours la dernière définition d'une valeur qui est prise en compte.

Question 2. Donnez les résultats de l'évaluation des expressions suivantes puis vérifiez vos réponses en utilisant l'interpréteur Caml :

```
let x = 1 in x + 1;;

let y = 2 in let x = y in x + y;;

let x = 1 in let x = x + 1 in x + 1;;

let x = 1 and y = 2 in let x = x + y in x + y;;
```

2 Fonctions

En Caml un programme est une suite de définitions de fonctions, suivie d'un appel a une fonction, déclanchant le calcul voulu. Il existe deux manières de définir des fonctions :

```
let f = function x -> x + 1;;
```

ou :

```
let f x = x + 1;;
```

Pour appliquer une fonction on donne simplement son nom et son argument :

```
f (1);;
f (2 * 2);;
let x = 1 in f (x);;
```

Lorsqu'on applique une fonction a une constante ou une variable il est possible de ne pas écrire les parenthèses :

```
f 1;;
let x = 1 in f x;;
```

par contre l'appel suivant ne calcule pas $f(2 \times 2)$ mais $f(2) \times 2$:

```
f 2 * 2;;
```

Il est aussi possible de définir des fonctions à plusieurs arguments :

```
let g = function x -> function y -> x + y;;
```

ou :

```
let g x y = x + y;;
```

on comprend alors vraiment l'intérêt de la deuxième notation. Cependant il ne faut pas oublier, ce qui est clair avec la première notation, que g est une fonction qui associe a x une fonction qui associe a y la somme de x et y. En d'autres termes, pour calculer $g(1,2)$ il faut en toute rigueur écrire :

```
(g 1) 2;;
```

Cependant, en Caml l'application est associative « à gauche ». C'est à dire que `f x y` est équivalent à `(f x) y`. Donc, pour calculer $g(1,2)$ on peut écrire :

```
g 1 2;;
```

Question 3. À partir de maintenant nous allons écrire nos programmes dans un éditeur de texte (emacs) pour éviter de devoir tout retaper à chaque erreur. Pour ne pas avoir à copier tous les programmes dans l'interpréteur, ce qui peut vite s'avérer laborieux, nous utiliserons un mode d'emacs (tuareg) proposant un interpréteur intégré. Pour lancer le mode tuareg il suffit normalement d'ouvrir un fichier .ml dans emacs. Si cela ne fonctionne pas on pourra utiliser la commande `alt+x tuareg-mode`. Une fois le mode tuareg en route on peut lancer l'interpréteur par la commande `ctrl-c ctrl-e`. Chaque fonction est ensuite interprétée en positionnant le curseur sur celle-ci et en tapant `ctrl-c ctrl-e`.

Pour en savoir plus sur l'utilisation d'emacs vous pouvez consulter la page suivante : http ://www.tuteurs.ens.fr/unix/editeurs/emacs.html ou bien le guide – en anglais – intégré à emacs (commande `ctrl-h t`).

Rentrez quelques fonctions (successeur, carré...) dans emacs puis testez les.

3 Filtrage

Le filtrage en Caml est en fait une analyse de cas, il se présente de la manière suivante :

```
let f = function
   |cas1 -> x
   |cas2 -> y
;;
```

ce qui correspond à : si l'argument de f est `cas1` retourner `x`, si c'est `cas2` retourner `y`. Par exemple, la fonction `egal_un`, qui teste si son argument vaut 1 ou non, peut s'écrire :

```
let egal_un = function
   |1 -> true
   |_ -> false
;;
```

Le signe `_` signifiant « dans tous les autres cas ». Il est important de noter que l'évaluation des cas se fait de haut en bas : si deux cas sont acceptables c'est le premier qui sera pris en compte. Par exemple, la fonction suivante n'est pas équivalente a `egal_un`, elle retourne toujours `false` :

```
let bidon = function
   |_ -> false
   |1 -> true
;;
```

Il est aussi possible d'écrire le filtrage de manière explicite par le mot clé `match with` :

```
let egal_un x =
   match x with
      |1 -> true
      |_ -> false
;;
```

On peut aussi utiliser le mot clé `when` pour réaliser des tests plus compliqués. Par exemple, au lieu de la fonction suivante (fonctionnant uniquement sur les entiers positifs) :

```
let sup_2 = function
   |1 -> false
   |2 -> false
   |_ -> true
;;
```

on peut écrire :

```
let sup_2 = function
   |x when x>2 -> true
   |_ -> false
;;
```

qui est plus courte et fonctionne sur tous les entiers. Le filtrage est un élément très important de Caml. Il sera utilisé dans bon nombre de fonctions que vous écrirez.

Question 4. Écrivez une fonction qui retourne la valeur absolue d'un entier.

4 Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle même. En raison de la portée des noms en Caml il n'est pas possible d'écrire :

```
let somme = function
   |0 -> 0
   |n -> n + somme (n - 1)
;;
```

il est nécessaire de spécifier que la fonction s'appelle elle même. Pour cela on utilise le mot clé `let rec` :

```
let rec somme = function
   |0 -> 0
   |n -> n + somme (n - 1)
;;
```

Il faut être très prudent lorsque l'on écrit des fonctions récursives, par exemple, la fonction `somme` ne doit surtout pas être appelée avec un argument négatif.

Question 5. Écrivez une fonction `factorielle`, qui calcule la factorielle d'un nombre n.

Question 6. Écrivez une fonction `decompte` qui affiche les nombres de n à 1 et une fonction `compte` qui affiche les nombres de 1 à n. Ces fonctions prennent en argument l'entier n. Pour cela vous pouvez utiliser les fonctions `print_int` et `print_newline` de Caml, qui affichent respectivement à l'écran un entier et une nouvelle ligne.

Question 7. Écrivez une fonction `hanoi` qui résoud le problème des tours de hanoi pour un nombre n de disques. Cette fonction prendra en arguments trois chaînes de caractères (les noms des plateaux) et un entier (le nombre de disques). Elle affichera à l'écran les mouvements à effectuer pour résoudre le problème.

En Caml les chaînes de caractères se représentent entre guillemets : `"a"` représente la chaîne « a ». On peut les afficher par la fonction `print_string`.

5 Listes

En Caml les listes sont une structure de données très utilisée. Elles sont faciles à manipuler. Dans cette partie du TP nous allons implémenter plusieurs opérations de base sur les listes. Ces opérations sont disponibles dans le langage mais il est utile de les avoir vues au moins une fois. On ne pourra donc utiliser dans un premier temps que `[]` (la liste vide) et `::` (l'opérateur ajoutant un élément en tête de liste).

Question 8. Écrivez une fonction `est_vide` qui retourne `true` si une liste est vide est `false` sinon.

Question 9. Écrivez une fonction `tete` qui retourne le premier élément d'une liste.

Question 10. Écrivez une fonction `queue` qui retourne la queue d'une liste, c'est à dire la liste à laquelle le premier élément a été retiré.

Question 11. Écrivez une fonction `concat` qui concatène deux listes.

Question 12. Écrivez une fonction `longueur` qui compte le nombre d'éléments dans une liste.

Question 13. Èrivez une fonction `applique` qui étant donnée une liste $e_1, e_2, e_3, \ldots$ et une fonction f retourne la liste $f(e_1), f(e_2), f(e_3), \ldots$. La fonction f doit être un argument de `applique`.

Question 14. Écrivez une fonction `retourne` qui retourne une liste.

Toutes ces fonctions (et bien d'autres) sont disponibles dans le module `list` de la bibliothèque standard de Caml.

6 Programmation impérative en Caml

Il est possible de programmer de manière impérative en Caml. Si beaucoup de choses s'écrivent beaucoup plus élégamment et de manière plus concise à l'aide du filtrage et des fonctions récursives il est tout de même plus simple d'écrire certaines fonctions de manière purement impérative. Les mêmes structures de contrôle sont disponibles en Caml que dans la plupart des langages impératifs :

```
if x >= 0 then ... else ...;;

for i=0 to 5 do ... done;;

for i=5 downto 0 do ... done;;

while i<5 do ... done;;
```

Il est possible aussi, grace au mot clé `ref`, d'utiliser des références (assimilables aux pointeurs en C). On peut déréférencer par le signe `!` et modifier la valeur associée à une référence par `:=` :

```
let i = ref 0 in
   while !i < 5 do
      ...
      i := !i + 1;
      ...
   done
;;
```

On peut bien sur construire des tableaux, soit directement :

```
let tab = [| 0; 0; 0 |];;
```

soit en utilisant la fonction `make_vect` :

```
let tab = make_vect 3 0;;
```

On accède aux cases d'un tableau de la manière suivante :

```
tab.(i);;
```

et on peut affecter des valeurs aux cases d'un tableau :

```
tab.(i) <- 5;;
```

De nombreuses autres fonctions sur les tableaux sont disponibles dans le module `array` de la bibliothèque standard de Caml.

Question 15. Écrivez de manière impérative les fonctions `factorielle`, `compte` et `decompte` vues précédemment.

Attention : de manière général il vaut mieux essayer d'éviter au maximum la programmation impérative.

7 Déclaration de nouveaux types

Il est possible en Caml de définir de nouveaux types. Pour cela on utilise le mot clé `type`. Il existe deux sortes de types définissables : les types somme et les types produit. Un type somme se construit de la manière suivante :

```
type couleur =
   | Blanc
   | Noir
;;
```

On peut alors donner un nom à des valeurs de ce nouveau type :

```
let c = Blanc;;
```

On peut aussi filtrer ce type :

```
let est_blanc = function
   |Blanc -> true
   |_ -> false
;;
```

On peut aussi avoir des constructeurs ayant un argument, par exemple :

```
type couleur =
   | Blanc
   | Noir
   | Autre of int
;;
```

On construit et filtre alors ainsi :

```
let c = Autre(3);;

let val_autre = function
   |Autre(x) -> x
   |_ -> -1
;;
```

Enfin on peut définir des types récursifs :

```
type couleur =
   | Blanc
   | Noir
   | Melange of couleur * couleur
;;
```

Un type produit (ou enregistrement) se définit ainsi :

```
type monome = { coefficient : int; degre : int };;
```

Pour construire une valeur d'un tel type on énumère ses caractéristiques :

```
let m = { coefficient = 1; degre = 0 };;
```

On peut accéder de manière indépendante aux différentes caractéristiques d'une valeur ayant un tel type :

```
m.coefficient;;

m.degre;;
```

On peut bien sur filtrer :

```
let degre_zero = function
   |{ coefficient = _; degre = 0 } -> true
   |_ -> false
;;
```

Enfin il est possible de définir des caractéristiques « modifiables »dans un type produit, on utilise pour cela le mot clé `mutable` :

```
type monome = { mutable coefficient : int; degre : int };;
```

On peut alors modifier ces caractéristiques pour une valeur donnée :

```
let m =  { coefficient = 1; degre = 0 };;

m.coefficient <- 2;;
```

Question 16. Définissez un type `complexe`, représentation rectangulaire d'un nombre complexe. Écrivez les fonctions `mon_module` et `mon_argument` permettant d'obtenir le module et l'argument d'un complexe.

8 Exercices complémentaires

On peut aussi définir des types polymorphes :

```
type 'a couple = 'a * 'a;;
```

Question 17. Définissez un type `'a pile` ainsi que les fonctions permettant d'agir sur des valeurs de ce type (`pile_vide`, `sommet`, `empiler`, `depiler`, `est_vide`).

Question 18. Écrivez une fonction qui réalise un tri par insertion d'une pile. Le type de cette fonction devra être `('a -> 'a -> bool) -> 'a pile -> 'a pile`.

Question 19. Définissez les types correspondant aux autres structures de données (prochainement) vues en cours (files, arbres...) et les opérations correspondantes.