Magistère STIC – Programmation 1.1

1. Introduction à Caml

18 septembre 2006

1 Documentation

Objective Caml (juste Caml pour les intimes) est un langage de programmation mature pour lequel il existe diverses sources d'information en ligne ou sur papier. Le site web de référence se trouve à l'adresse `http://caml.inria.fr/`, vous y trouverez en particulier le guide de référence[1] qui décrit précisément le langage est ses bibliothèques standard, ainsi qu'une liste de documents[2] dont tutoriels, références de livres, etc. Je vous encourage à consulter tout ça.

2 Prise en main

Il y a deux façons d'utiliser Caml : l'interpréteur et le compilateur. L'interpréteur lit des expressions et des déclarations de fonctions, les évalue immédiatement, et affiche les résultats (en anglais, on parle de boucle *read-eval-print*). On peut utiliser l'interpréteur directement dans un terminal en tapant la commande « `ocaml` » :

```
$ ocaml
        Objective Caml version 3.09.2

# 2 + 2;;
- : int = 4
# exit 0;;
$
```

Un programme en Caml est consitué d'un ou plusieurs fichiers source dont le nom est de la forme « `toto.ml` ». Supposons qu'on ait un le fichier suivant :

```
(* toto.ml - programme d'exemple *)
print_int (2 + 2) ;;
print_newline ()
```

Un tel programme peut être exécuté directement par l'interpréteur en tapant la commande « `ocaml toto.ml` » :

[1] `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`
[2] `http://caml.inria.fr/resources/doc/index.fr.html`

```
$ ocaml toto.ml
4
$
```

Le programme peut aussi être compilé en un programme exécutable en utilisant « `ocamlc` » :

```
$ ocamlc toto.ml -o toto
$ ./toto
4
$
```

L'interpréteur peut aussi être utilisé depuis un éditeur de texte comme Emacs qui permet d'évaluer des expressions présentes dans le fichier en cours d'édition. En principe, le mode d'édition pour Caml est lancé automatiquement lorsque vousouvrez un fichier dont le nom se termine en « `.ml` ». La combinaison de touches Ctrl-C puis Ctrl-S sert à lancer Caml, puis en cours d'édition Ctrl-X Ctrl-E évalue l'expression sur laquelle se trouve le curseur.

Les gens forts en marketing disent que Caml est un langage multi-paradigmes, car il permet de programmer en styles fonctionnel, impératif et objet. La base du langage, que l'on nomme Core-ML, est un langage fonctionnel, et c'est ce que nous allons étudier aujourd'hui. Pour le début, utilisez l'interpéteur en mode interactif.[3] Et ne me croyez pas sur parole, n'hésitez pas à taper les exemples.

3 Expressions

Les opérations arithmétiques sur les entiers n'ont rien d'exceptionnel. Notez simplement que toute expression donnée à évaluer à Caml doit être terminée par « `;;` » :

```
# 1 + 2 * 3 ;;
- : int = 7
```

L'interpréteur répond le type de l'expression et sa valeur. La même chose s'applique aussi aux nombres à virgule flottante :

```
# 1. +. 2.5 *. 3.78 ;;
- : float = 10.45
```

Note : les opérateurs `+`, `-`, `*`, `/` ne s'appliquent qu'aux entiers, et les opérateurs `+.`, `-.`, `*.`, `/.` ne s'appliquent qu'aux flottants. Caml étant un langage *fortement typé*, il n'y a aucune conversion automatique entre entiers et flottants :

```
# 1 + 2. ;;
This expression has type float but is here used with type int
```

L'application de fonctions se fait en apposant la fonction et ses arguments :

[3] On se rend vite compte que l'interpréteur ne permet pas facilement de corriger ce que l'on tape. Il est plus confortable d'utiliser un éditeur de ligne de commande comme « `rlwrap` » ou « `ledit` », en lançant l'intepréteur en disant « `rlwrap ocaml` ». Ou alors, faites du copier-coller. Ou alors, utilisez Emacs, lequel sait interagir avec un interpréteur. Ou alors, ne faites jamais de fautes de frappe.

```
# (sqrt 3. +. 1.) /. 2. ;;
- : float = 1.3660254037844386
```

Exercice 1 Comprendre les lignes suivantes, les taper, et comprendre les réponses de Caml :

```
"t'as le bon" ^ "jour d'Alfred" ;;
3 < 4 ;;
if 78. >= 15. then 'a' else 'b' ;;
true && false ;;
char_of_int 65 ;;
(1.5, "blop", 3 + 8) ;;
sqrt ;;
() ;;
```

4 Déclarations et fonctions

La construction **let** permet de nommer des expressions :

```
# let x = 22 + 47 in x / 5 ;;
- : int = 13
```

Sans la partie **in**, cette construction permet de nommer une expression pour le reste de la session :

```
# let answer = 6 * 7 ;;
val answer : int = 42
# answer + 8 ;;
- : int = 50
```

La construction **let** permet aussi de définir des fonctions :

```
# let double x = x + x ;;
val double : int -> int = <fun>
# double 6 ;;
- : int = 12
```

Pour définir une fonction récursive, on doit employer le mot-clé **rec** :

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

Au passage, essayez de taper la derinère ligne sans les parenthèses, c'est-à-dire « `plus -1 3` » et expliquez le résultat.

Exercice 2 – *Conditionnelles*

Pour partager une somme entre deux personnesm on demande à chacune d'entre elles combien elle prétend recevoir. Soit S la somme à partager et soient x_1 et x_2 les sommes réclamées.

– si $x_1 + x_2 \leqslant S$, chacun reçoit ce qu'il réclame ;
– si $S \leqslant \min(x_1, x_2)$, chacun reçoit la moitié de la somme ;
– si $x_1 \leqslant S \leqslant x_2$, la personne 1 reçoit $x_1/2$ et l'autre le reste ;
– si $x_2 \leqslant S \leqslant x_1$, la personne 2 reçoit $x_2/2$ et l'autre le reste ;
– sinon la personne 1 reçoit $(S + x_1 - x_2)/2$ et l'autre reçoit $(S + x_2 - x_1)/2$.

Écrire une fonction qui prend en argument la somme S et le couple (x_1, x_2) et renvoie un couple d'entiers correpondant à la somme attribuée à chaque personne.

Exercice 3 – *Liaison statique*

Quel est la réponse à la dernière ligne de la session suivante ?

```
# let a = 5 ;;
val a : int = 5
# let f x = x + a ;;
val f : int -> int = <fun>
# f 8 ;;
- : int = 13
# let a = 2 ;;
val a : int = 2
# f 6 ;;
```

Exercice 4 – *Un classique*

Écrire une fonction `fibo` telle que `fibo n` renvoie le n-ième élément de la suite de Fibonacci (dont je ne vous ferai pas l'affront de rappeler la définition).

Retenez bien le slogan : *les fonctions sont des valeurs comme les autres*. Elles peuvent donc être passées en arguments à d'autres fonctions et utilisées comme valeurs de retour pour d'autres fonctions. Il est possible également d'écrire une fonction sans lui donner de nom :

```
# fun x -> x + 1 ;;
- : int -> int = <fun>
# (fun x -> x * 3) 8 ;;
- : int = 24
```

Exercice 5 – *Compositions*

Définir une fonction `compose` telle que `compose f g` renvoie la fonction $f \circ g$. En déduire une fonction `itère` telle que `itère n f` renvoie la fonction f^n, qui à x associe le résultat de n applications successives de f à x. En utilisant cette fonction définir une fonction `triple` qui prend en argument une fonction f et renvoie $f \circ f \circ f$.

5 Listes

La liste est une structure de donnée fondamentale de Caml, et elle bénéficie d'une syntaxe particulière. Une liste ne peut contenir que des éléments de même type.

```
# let lst = [1; 8; 9] ;;
val lst : int list = [1; 8; 9]
# 3 :: lst ;;
- : int list = [3; 1; 8; 9]
# "pif" :: lst ;;
This expression has type int list but is here used with type string list
```

La bibliothèque standard contient un certain nombre de fonctions pour manipuler les listes, dans un module judicieusement nommé `List`. Nous aurons l'occasion de revenir sur le système de modules de Caml, pour le moment retenez simplement qu'une fonction `foo` d'un module `Bar` peut être utilisée en employant la syntaxe `Bar.foo`. Alternativement, la déclaration `open Bar` rend directement accessibles les fonctions de ce module, de sorte aue `foo` devient équivalent à `Bar.foo`.

```
# List.length lst ;;
- : int = 3
# List.rev lst ;;
- : int list = [9; 8; 1]
# lst @ lst ;;
- : int list = [1; 8; 9; 1; 8; 9]
```

Le filtrage (en anglais on parle de *pattern matching*) est une technique typique des langages de programmation fonctionnels. Il permet de définir une fonction par cas selon la forme de la donnée qu'elle reçoit. Par exemple, pour définir une fonction qui calcule la longueur d'une liste, on peut procéder de la façon suivante :

```
let rec length list =
  match list with
  | [] -> 0
  | _ :: queue -> 1 + length queue
```

Le motif `_` sert à accepter n'importe quelle valeur, on l'utilise ici parce que la valeur de la tête de la liste n'est pas prise en compte.

Exercice 6 – *Filtrages*

1. Définir une fonction `est_vide` qui renvoie `true` si et seulement si la liste passée en argument est vide.
2. Définir les fonctions `head` et `tail` qui renvoient respectivement la tête et la queue d'une liste.

   Note : Caml fournit une fonction `failwith` pour utiliser dans les cas d'erreur, par exemple :

   ```
   let div x y =
     if y = 0 then failwith "Division par zéro" else x / y ;;
   ```

3. Définir une fonction `map` qui, appliquée à une fonction f et une liste $[a_1; \ldots; a_n]$, renvoie la liste $[f(a_1); \ldots; f(a_n)]$.
4. Définir une fonction `un_sur_deux` qui prend une liste ℓ en argument et renvoie la liste composée des éléments de ℓ d'indice pair.
5. Définir une fonction `premiers` qui renvoie la liste des premiers éléments des éléments non vides d'une liste de listes. Par exemple, `premiers [[1;2;3]; [4]; []; [6;7]]` doit renvoyer `[1;4;6]`.

Le filtrage peut s'appliquer à n'importe quel type construit, dont les listes et les n-uplets, ainsi qu'aux types de base (mais un motif ne peut contenir que des constantes). On peut par exemple écrire :

```
let rec ackerman = function
  | (0, n) -> n + 1
  | (m, 0) -> ackerman (m - 1, 1)
  | (m, n) -> ackerman (m - 1, ackerman (m, n - 1))
```

La notation **`function`** est équivalente à **`fun`** `x ->` **`match`** `x` **`with`**. Les motifs peuvent aussi être utilisés avec **`let`** lorsqu'il n'y a qu'un seul cas de filtrage, ainsi **`let`** `(x,y) = f 12` **`in`** `a` est équivalent **`match`** `f 12` **`with`** `(x,y) -> a`.

Exercice 7 – *Programmation combinatoire*

1. La fonction `List.map` prend en arguments une fonction f et une liste ℓ et elle renvoie la liste obtenue en appliquant f à chaque élément de ℓ. Définir une fonction `double_list` qui prend une liste d'entiers et multiplie chaque élément par 2.
2. Lire la définition de la fonction `List.fold_left`, la comprendre, et en déduire une fonction `produit` qui prend en argument une liste de flottants et renvoie le produit de ses éléments.
3. La fonction `List.concat` prend en argument une liste de listes d'éléments du même type et renvoie la concaténation des éléments de cette liste. En déduire une fonction `bégaie` qui prend une liste en argument et renvoie la liste obtenue en dupliquant chaque élément.