Introduction à Caml (1)

1 Questions fréquemment posées au sujet de Caml

Quelques extraits de l'ancienne FAQ Caml, que l'on peut trouver à l'adresse : `http://caml.inria.fr/pub/old_caml_site/FAQ/FAQ_DEBUTANT-fra.html`

1.1 Qu'est-ce que Caml ?

Caml est un langage de programmation. C'est un langage fonctionnel car les briques de base des programmes sont les fonctions. C'est un langage fortement typé, ce qui signifie que les objets manipulés appartiennent à un ensemble identifié par un nom qu'on appelle son type. En Caml, les types sont gérés et manipulés par la machine, sans intervention de l'utilisateur (types synthétisés).
Le langage est disponible sur à peu près toutes les machines Unix, sur PC et sur Mac. Une utilisation confortable nécessite 2 Mo de mémoire libre et 2 Mo de mémoire sur disque dur.

1.2 Que signifie le nom « Caml » ?

« Caml » est un acronyme en langue anglaise : « Categorical Abstract Machine Language ». La CAM est une machine abstraite capable de définir et d'exécuter les fonctions, et qui est issue de considérations théoriques sur les relations entre la théorie des catégories et le lambda-calcul. Le premier compilateur du langage générait du code pour cette machine abstraite (en 1984). La deuxième filiation de ce nom est ML (acronyme pour Meta Language) : Caml est aussi issu de ce langage de programmation crée par Robin Milner en 1978, qui permet la reconnaissance de motifs et servait à programmer les tactiques de preuves dans le système de preuves LCF.

1.3 Est-ce un langage compilé ou interprété ?

Caml est un langage compilé. Cependant tous les systèmes Caml (on entend par « système » l'ensemble compilateur+bibliothèques associées) proposent une boucle d'interaction « toplevel », qui ressemble à s'y méprendre à un interprète. En effet dans le système interactif, l'utilisateur tape des morceaux de programmes (des « phrases » Caml) qui sont traitées instantanément par le système, qui les compile, les exécute et écrit les résultats à la volée.

1.4 Où trouver de la documentation ?

`http://caml.inria.fr/caml-light/index.fr.html`

2 Généralités

2.1 Constantes « globales »

Les constantes sont définies en Caml à l'aide de la construction `let` *identificateur* = *expression*, qui lie le nom *identificateur* à au résultat de l'évaluation de *expression* :

```
# let x=2;;
x : int = 2
# x*3;;
- : int = 6
```

L'identificateur est alors lié à la valeur donnée ; il est impossible de modifier cette valeur sans redéfinir une constante portant le même nom. Les variables de Caml, comme les variables mathèmatiques, ne peuvent pas changer (par opposition aux variables des langages impératifs).

Remarque

Ces définitions sont « statiques », ie une redéfinition n'a pas d'effet rétroactif :

```
# let a=1;;
a : int = 1
# let b=a+1;;
b : int = 2
# let a=2;;
a : int = 2
# print_int b;;
2- : unit = ()
```

Un exemple utilisant une fonction :

```
# let a=1;;
a : int = 1
# let f=function x -> x+a;;
f : int -> int = <fun>
# let a=10;;
a : int = 10
# f 2;;
- : int = 3
```

Liaisons locales

La construction utilisée est `let` ... `in` ... .

```
# let a=1 in a+6;;
- : int = 7
```

Ces liaisons n'ont de portée que dans l'expression qui suit :

```
# let a=5;;
a : int = 5
# let a=2 in 3*a;;
- : int = 6
# print_int a;;
5- : unit = ()
```

On peut également utiliser la construction ... `where` ... :

```
# 12-a where a=3;;
- : int = 9
```

Liaisons locales simultanées

On utilise `and` :

```
# let a=1 and b=2 in a+4*b;;
- : int = 9
```

Il faut toutefois que ces liaisons soient indépendantes. Dans le cas contraire, on imbriquera deux `let` ... `in` :

```
# let a=1 and b=a+2 in 2*b-a;;
Entree interactive:
> let a=1 and b=a+2 in 2*b-a;;
>                 ^
L'identificateur a n'est pas defini.
# let a=1 in
  let b=a+2 in
  2*b-a;;
- : int = 5
```

2.2 Priorité des opérateurs et parenthésage

En Caml comme dans la plupart des langages, les opérateurs possèdent une priorité. Par exemple, la multiplication est prioritaire par rapport à l'addition :

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

L'ordre d'évaluation peut toutefois être modifié par l'adjonction de parenthèses :

```
# (1+2)*3;;
- : int = 9
```

Par ailleurs, le parenthésage « suit les règles mathématiques des fonctions trigonométriques » (Pierre Weis, l'un des auteurs de Caml). Ainsi, si f désigne une fonction, on pourra écrire `f x` pour `f(x)` (tout comme $\sin x$ signifie $\sin(x)$) et par exemple `f x - 1` signifie `f(x)-1` (tout comme $\sin x - 1$ signifie $\sin(x) - 1$) :

```
# let f = function x->x*x in f 9 - 1;;
- : int = 80
```

2.3 Commentaires

Les commentaires sont placés entre les symboles `(*` et `*)` (éventuellement sur plusieurs lignes) et sont alors ignorés par le compilateur. Les programmes étant en général destinés à être lus par plusieurs personnes, ne pas hésiter à les commenter ...

3 Types usuels

3.1 Les entiers (int)

Sur les systèmes 32 bits, les « entiers » de Caml sont compris entre -2^{30} et $2^{30}-1$ (et entre -2^{62} et $2^{62}-1$ sur machines 64 bits pour des versions récentes de (O)Caml), ce qui peut conduire à certaines déconvenues :

```
# 123456*12345;;
- : int = -623419328
```

Opérations sur les entiers

`+` (addition), `-` (soustraction), `*` (multiplication), `/` (quotient de la division euclidienne) et `mod` (reste de la division euclidienne). Les valeurs retournées par ces fonctions sont des entiers. Par exemple :

```
# 31/5;;
- : int = 6
# 31 mod 5;;
- : int = 1
```

Fonctions de comparaison entre entiers

`<`, `>` (inégalités strictes), `<=`, `>=` (inégalités larges), `<>` (pour $\neq$). Les valeurs retournées par ces fonctions sont des booléens.

```
# 1+1>=2-1;;
- : bool = true
```

3.2 Les réels (float)

La représentation des réels en Caml est semblable à celle des calculatrices (mantisse et puissance de 10 éventuelle, notée à l'aide de `e` ou `E`).

```
# 1.e5;;
- : float = 100000.0
```

Opérations sur les réels

`+.` (addition), `-.` (soustraction), `*.` (multiplication), `/.` (division), et les fonctions usuelles telles que `exp`, `log`, `sin`, etc.

```
# 2.e4/.24.;;
- : float = 833.333333333
```

Conversion réels/entiers

On dispose des fonctions `int_of_float:float->int` et `float_of_int:int->float`.

```
# float_of_int 12;;
- : float = 12.0
# int_of_float 2.3;;
- : int = 2
# int_of_float (-. 2.3);;
- : int = -2
```

3.3 Les booléens (bool)

Il y a deux booléens de base : `true` et `false`.

```
# 1+1>=2-1;;
- : bool = true
```

Opérations sur les booléens

`&` ou `&&` (pour le « et »), `or` ou `||` (pour le « ou »), `not` (négation).

```
# true or (not true);;
- : bool = true
# true && false;;
- : bool = false
```

3.4 Le type unit

Il existe des fonctions Caml ne retournant aucun résultat (on dit qu'elles procèdent par « effets de bords »). C'est en particulier le cas des fonctions d'impression `print_int`, `print_float`, `print_string`, etc.
Tout étant typé en Caml, un type particulier leur est assigné : `unit`, dont le seul élément est noté `()`. Plusieurs fonctions de ce type peuvent être enchaînées.

```
#print_int 2;;
2- : unit = ()
#let f = function () -> print_newline();;
f : unit -> unit = <fun>
# f ();;

- : unit = ()
# print_int 2 ; 1+3;;
2- : int = 4
```

3.5 Les caractères (char)

Ils sont notés entre accents graves et permettent principalement d'afficher des codes ASCII. On dispose pour cela des fonctions `char_of_int:int->char` et `int_of_char:char->int` :

```
# int_of_char `a`;;
- : int = 97
# char_of_int 55;;
- : char = `7`
```

3.6 Les chaînes de caractères (string)

Elles se notent à l'aide de guillemets (la chaîne vide est donc représentée par `""`) :

```
# let a="Bonjour";;
a : string = "Bonjour"
```

Opérations sur les chaînes

On peut concaténer deux chaînes à l'aide de l'opérateur `^` :

```
# let b="tout le monde !";;
b : string = "tout le monde !"
# a ^ b;;
- : string = "Bonjour tout le monde !"
```

La fonction `string_length:string->int` donne par ailleurs la longueur d'une chaîne :

```
# string_length b;;
- : int = 15
```

On a également accès au i-ème caractère d'une chaîne (attention, l'indice du premier caractère est 0) :

```
# b.[5];;
- : char = ‘l‘
# b.[string_length b];;
Exception non rattrapee: Invalid_argument "nth_char"
```

Enfin, on peut transformer un caractère d'une chaîne :

```
# b.[8]<-‘b‘;;
- : unit = ()
# print_string b;;
tout le bonde !- : unit = ()
```

Attention aux modifications dans les chaînes, elles se somportent comme des vecteurs :

```
# let c1="douche";;
c1 : string = "douche"
# let c2=c1;;
c2 : string = "douche"
# c1.[0]<-‘m‘;;
- : unit = ()
# print_string c2;
mouche- : unit = ()
```

Conversions chaînes/entiers ou réels

On dispose des fonctions `string_of_int`, `string_of_float`, `int_of_string`, `float_of_string` :

```
# string_of_float 20.;;
- : string = "20.0"
# float_of_string "2";;
- : float = 2.0
```

3.7 Les tableaux (vect)

En Caml, les tableaux (ou vecteurs) sont de la forme `[|`e_0`;`e_1`;...;`e_{n-1}`|]` où les e_i sont tous du même type. Le tableau vide est noté `[||]`.
Un tableau peut être considéré comme un ensemble de cases mémoires de taille fixe (calculable à l'aide de la fonction `vect_length:'a vect->int`).

```
# let tab=[|1.;2.;5.|];;
tab : float vect = [|1.0; 2.0; 5.0|]
```

```
# vect_length tab;;
- : int = 3
```

Les indices des éléments sont compris entre 0 et $n-1$, où n désigne la longueur du tableau :

```
# tab.(2)<-8.;;
- : unit = ()
# tab;;
- : float vect = [|1.0; 2.0; 8.0|]
```

La commande `make_vect:int -> 'a -> 'a vect` permet de créer un tableau de longueur donnée rempli à l'aide de l'élément fourni en argument :

```
# let t = make_vect 5 "toto";;
t : string vect = [|"toto"; "toto"; "toto"; "toto"; "toto" |]
```

Les vecteurs sont toujours passés par référence aux fonctions, donc sont modifiés par les opérations locales à ces fonctions :

```
# let t=[|0;1;2|];;
t : int vect = [|0; 1; 2|]
# let f = function t -> t.(0)<-17;;
f : int vect -> unit = <fun>
# f(t);;
- : unit = ()
# t;;
- : int vect = [|17; 1; 2|]
```

De même, créer une liaison revient à partager un même espace mémoire, avec des conséquences parfois déroutantes :

```
# let t1=t;;
t1 : int vect = [|17; 1; 2|]
# t.(1)<-24;;
- : unit = ()
# t1;;
- : int vect = [|17; 24; 2|]
```

3.8 Les références

Les références servent à définir des variables (par opposition aux constantes) en Caml. Une référence est une plage mémoire de taille fixe ; elle peut être vue comme une boîte pourvue d'une étiquette et pouvant contenir des objets. La définition d'une référence s'opère à l'aide de la construction `let` *identificateur* `= ref` *contenu* ; le contenu est alors accessible grâce à l'opérateur `!` et modifiable avec l'opérateur `:=`. À noter que l'on ne peut changer le type du contenu :

```
# let a=ref 1;;
a : int ref = ref 1
```

```
# a;; (* etiquette de la boite *)
- : int ref = ref 1
# !a;; (* contenu de la boite *)
- : int = 1
# a:=2;;
- : unit = ()
# !a;;
- : int = 2
# a:=0.0;;
Entree interactive:
> a:=0.0;;
>    ^^^
Cette expression est de type float
mais est utilisee avec le type int.
```

Une commande utile : `incr` (et son pendant `decr`) :

```
# incr;;
- : int ref -> unit = <fun>
# let x = ref 0;;
x : int ref = ref 0
# incr x;;
- : unit = ()
# x;;
- : int ref = ref 1
```

3.9 Les produits cartésiens

À partir de types préexistants, on peut définir de nouveaux types (couples voire n-uplets, correspondant à un produit cartésien en mathématiques) :

```
# (1,2);;
- : int * int = 1, 2
```

Noter le caractère `*` qui désigne un produit cartésien. Les parenthèses sont facultatives, et il n'est pas nécessaire que toutes les composantes soient du même type :

```
# 1,2.,"bonjour",false;;
- : int * float * string * bool = 1, 2.0, "bonjour", false
```

Le formalisme précédent permet de définir simultanément plusieurs constantes :

```
# let x,y,z = 1,1.2,"toto";;
x : int = 1
y : float = 1.2
z : string = "toto"
```

Sur les couples, les fonctions `fst:'a*'b->'a` (comme « first ») et `snd:'a*'b->'b` (comme « second ») permettent d'accéder respectivement à la première et la seconde composante :

```
# let a = (1.,"toto");;
a : float * string = 1.0, "toto"
# snd a;;
- : string = "toto"
```

3.10 Autres types

D'autres types prédéfinis (comme les listes) seront vus ultérieurement, ainsi que les manières de définir ses propres types.

4 Éléments de programmation impérative

4.1 Tests

La syntaxe est des plus classiques : `if` *condition* `then` *bloc d'instructions 1* `else` *bloc d'instructions 2*. Les deux blocs d'instructions doivent nécessairement fournir des résultats de même type. La partie « `else` bloc d'instructions 2 » est facultative, mais dans ce cas le résultat de « bloc d'instructions 1 » doit être de type `unit`.

```
# if 2<3 then 4 else 5;;
- : int = 4
# if 2<3 then 4;;
Entree interactive:
> if 2<3 then 4;;
>             ^
Cette expression est de type int,
mais est utilisee avec le type unit.
```

Toute suite d'instructions encadrée par `begin` et `end` est considérée comme une instruction unique par le compilateur (`begin`...`end` joue ainsi le rôle d'une paire de parenthèses) :

```
#let f=function
  x->if x>0 then
    begin
      print_string "positif !";
      print_newline()
    end
  else
    begin
      print_string "negatif !";
      print_newline()
```

```
    end;;
f : int -> unit = <fun>
```

— À noter que dans un bloc d'instructions, on peut « enchaîner » les effets de bord en les séparant par de simples points-virgules. On peut également effectuer plusieurs effets de bord, puis un calcul dont on renvoie le résultat, comme dans l'exemple suivant :

```
# let f x =
    let t = make_vect 5 0 in
    t.(0)<-2;
    print_string "OK !";
    2*x+1;;
f : int -> int = <fun>
```

Il n'est par contre pas permis d'enchaîner des calculs en séparant par des points-virgules :

```
# let a=1;let b=2;;
Toplevel input:
> let a=1;let b=2;;
>                 ^^
Erreur de syntaxe.
```

— En l'absence de `else`, le `then` ne porte que sur la première instruction qui le suit. Si plusieurs instructions doivent être exécutées lorsque la condition de test est réalisée, elles doivent être regroupées par `begin...end` ou une paire de parenthèses.

4.2 Boucles inconditionnelles

Syntaxe : `for` *variable* `=` *valeur initiale* `to` *valeur finale* `do` *bloc d'instructions* `done`
Le pas d'incrément est nécessairement égal à 1 (il peut toutefois être fixé à -1 en remplaçant `to` par `downto`). Si la valeur finale est strictement inférieure à la valeur initiale, le bloc d'instructions n'est pas exécuté ; ces deux valeurs sont nécessairement de type entier.

```
# for i=1 to 3 do
    print_int i;
    print_newline();
  done;;
1
2
3
- : unit = ()
```

4.3 Boucles conditionnelles

Syntaxe : `while` *condition* `do` *bloc d'instructions* `done;`.
Exemple : un entier x étant donné, la fonction suivante calcule le plus petit entier n tel que $2^n \geq x$:

```
# let f x =
    let puiss2 = ref 1 and exposant = ref 0 in
    while !puiss2 < x do
        puiss2 := !puiss2 * 2;
        exposant := !exposant + 1;
    done;
    !exposant;;
f : int -> int = <fun>
# f 10;;
- : int = 4
```

5 Introduction aux fonctions

La définition des fonctions en Caml est calquée sur la formulation mathématique usuelle :

```
# let f = function x->x+1;;
f : int -> int = <fun>
# f(2);;
- : int = 3
```

On peut également écrire :

```
# let f x = x +. 1.;;
f : float -> float = <fun>
```

Il est également possible de définir des fonctions à plusieurs arguments, ou agissant sur des couples :

```
# let f x y = x+y;;
f : int -> int -> int = <fun>
# f 2 3;;
- : int = 5
# let f(x,y) = x +. float_of_int y;;
f : float * int -> float = <fun>
```

Les fonctions seront traitées de manière plus approfondie dans le chapitre suivant ...