

CHAPITRE

CAML pour l' impatient



Un tutoriel très très léger pour débiter avec OCAML

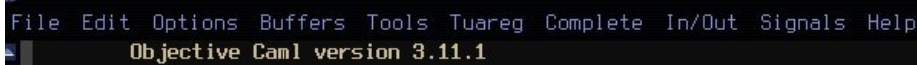
1 Installation et utilisation

Pour tous les O.S. il existe des distributions de CAML *clé-en-main* sur le page de Jean Mouric : <http://pagesperso-orange.fr/jean.mouric/>.

Pour les utilisateurs de Emacs, vous pouvez charger le mode tuareg :

<http://www.emacswiki.org/emacs/TuaregMode>.

CAML sous Emacs avec le mode tuareg :

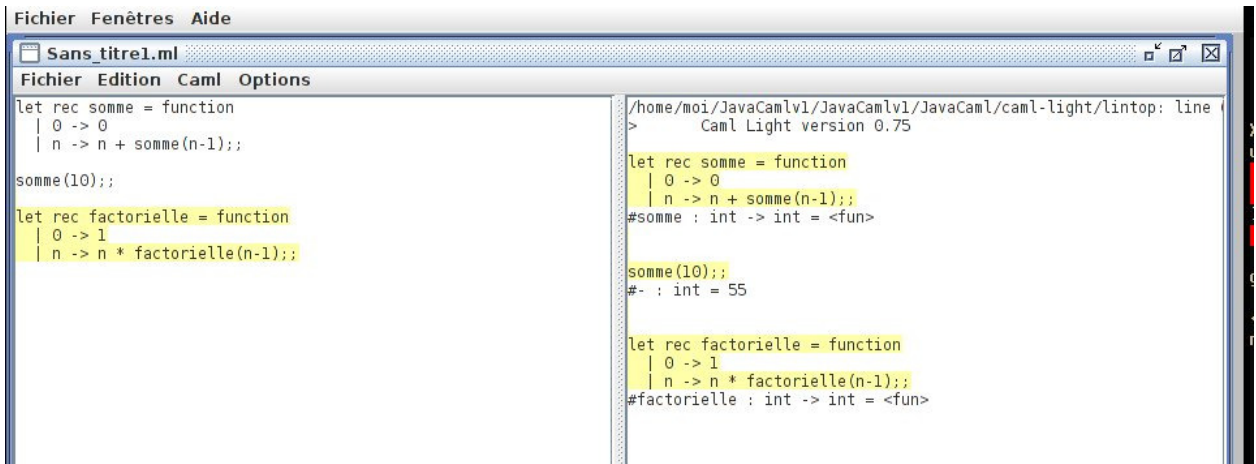


```
# let rec somme = function
| 0 -> 0
| n -> n + somme(n-1);;
val somme : int -> int = <fun>

# somme(10);;
- : int = 55

# let rec factorielle = function
| 0 -> 1
| n -> n * factorielle(n-1);;
```

CAML sous JavaCaml :



2 Quelques ressources électroniques

- Une passionnante introduction à OCAML par un de ses papas...
<http://caml.inria.fr/pub/distrib/books/llc.pdf>
- Le manuel de référence de CAML par Xavier LEROY et Pierre WEIS :

<http://caml.inria.fr/pub/distrib/books/manuel-cl.pdf>

- Une introduction à CAML :
http://fr.wikibooks.org/wiki/Objective_Caml
- Ouvrage de référence CAML :
<http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/>
- Ressources CAML :
<http://caml.inria.fr/resources/index.fr.html>
- Un tutoriel en français :
http://www.ocaml-tutorial.org/tutoriel_objective_caml

3

Les nombres

Nous utiliserons dans tout le document la version basique de CAML (en fait OCAML (c'est-à-dire CAML avec des modules pour faire de la programmation orientée objet) ou CAML Light (c'est la version de base que nous utiliserons)) sans charger de modules complémentaires sauf éventuellement pour créer des graphiques ou travailler sur certaines listes.

Nous travaillerons en mode *toplevel* c'est-à-dire que nous compilerons automatiquement de manière interactive. Pour se repérer, ce que nous taperons sera précédé d'un # et ce que renverra CAML sera précédé le plus souvent d'un -.

On peut travailler avec des entiers :

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

Vous remarquerez que CAML répond que le résultat de 1+2 est un entier (**int**) égal à 3. En effet, CAML est adepte de l'*inférence de type*, c'est-à-dire qu'il devine quel type de variable vous utilisez selon ce que vous avez tapé. Nous en reparlerons plus bas.

Les priorités des opérations sont respectées. En cas d'ambiguïté, un parenthésage implicite à gauche est adopté :

```
# 1+2*3;;
- : int = 7
# 1-2+3;;
- : int = 2
# 1-(2+3);;
- : int = -4
# (1+2)*3;;
- : int = 9
```

La division renvoie le quotient entier bien sûr :

```
# 11/3;;
- : int = 3
# 11/3*2;;
- : int = 6
# 11/(3*2);;
- : int = 1
```

On peut utiliser **mod** pour le reste entier :

```
# 7 mod 2;;
```

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

Enfin, les entiers sont de base compris entre -2^{31} et $2^{31} - 1$ sur un processeur 32 bits et entre -2^{63} et $2^{63} - 1$ sur un processeur 64 bits.

Les nombres non entiers sont de type *float*. Ils sont représentés en interne par deux entiers : une *mantisse* m et un exposant n tel que le nombre flottant soit $m \times 2^n$. En externe, il apparaît sous forme décimale, le séparateur étant le point.

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

Attention alors aux opérations :

```
# 3.14 + 2;;
Characters 0-4:
  3.14 + 2;;
  ^^^^
Error: This expression has type float but an expression was expected of
      type int
```

Cela indique que vous avez utilisé l'addition des entiers pour ajouter un entier à un flottant. CAML en effet n'effectue pas de conversion implicite et demande que celle-ci soit explicite. Cela peut apparaître comme contraignant mais permet l'inférence de type qui évite nombre de « bugs ».

Les opérations arithmétiques sur les entiers doivent donc être suivies d'un point :

```
# 1. +. 2.1 ;;
- : float = 3.1
# 1.2 +. 2.1 ;;
- : float = 3.3
# 1./2.;;
- : float = 0.5
# 1.5e-5 *. 100. ;;
- : float = 0.0015
# sqrt(2.);;
- : float = 1.41421356237309515
# 3.**2.;;
- : float = 9.
# log(2.);;
- : float = 0.693147180559945286
# exp(1.);;
- : float = 2.71828182845904509
# cos(0.);;
- : float = 1.
# cos(2.*.atan(1.));;
- : float = 6.12303176911188629e-17
# sin(2.*.atan(1.));;
- : float = 1.
```

Il existe des moyens de convertir un entier en flottant :

```
# float(1) +. 3.1;;
- : float = 4.1
# float 1 +. 3.1;;
- : float = 4.1
```

et inversement :

```
# int_of_float(sqrt(2.));;
- : int = 1
```

Il ne faut pas confondre `int_of_float` et `floor`

```
# floor(2.1);;
- : float = 2.
# int_of_float(2.1);;
- : int = 2
# floor(-2.1);;
- : float = -3.
# int_of_float(-2.1);;
- : int = -2
```

4

Les autres types de base

4.1 Les booléens

Ils sont bien sûr au nombre de deux : `true` et `false`. Nous en aurons besoin pour les tests. Les fonctions de comparaison renvoient un booléen. On peut combiner des booléens avec `not`, `&` et `or` :

```
# 3>2;;
- : bool = true
# 3=2;;
- : bool = false
# (3>2) & (3=2);;
- : bool = false
# (3>2) or (3=2);;
- : bool = true
# (3>2) & not(3=2);;
- : bool = true
# (3>2) & not(3=2) & (0<1 or 1>0);;
- : bool = true
```

4.2 Les chaînes de caractères

En anglais : *string*. On les entoure de guillemets `vb+"` et on les concatène avec `^` :

```
# "Tralala" ^ "pouet pouet";;
- : string = "Tralala pouet pouet"
```

4 3 Les caractères

En CAML : *char*. On les utilisera surtout pour la cryptographie. Ils sont entrés entre accents aigus :

```
# 'a';;
- : char = 'a'
```

5 Listes et tableaux

Il existe un type tableau (ou vecteur) comme dans de nombreux langages. Sa longueur est fixée lors de sa création ce qui permet d'avoir accès à ses composantes :

```
# let t = [|1;2;3|] ;;
val t : int array = [|1; 2; 3|]
# t.(2) ;;
- : int = 3
```

Il existe un type liste qui est chaînée et dont le longueur est donc dynamique mais on n'a accès qu'à son premier élément comme nous le verrons plus tard. On a cependant un opérateur binaire de concaténation @ :

```
# let liste = [1;2;3] ;;
val liste : int list = [1; 2; 3]
# 0::liste ;;
- : int list = [0; 1; 2; 3]
# [1;2] @ [3;4];;
- : int list = [1; 2; 3; 4]
```

Attention ! L'inférence de type a ses petites manies :

```
# [1; "deux" ; 3.0];;
Characters 4-10:
[1; "deux" ; 3.0];;
      ^^^^^^
Error: This expression has type string but an expression was expected of
      type
        int
# [1; 2 ; 3.0];;
Characters 8-11:
[1; 2 ; 3.0];;
      ^^^
Error: This expression has type float but an expression was expected of
      type
        int
```

6 Création de types

On n'insistera pas sur ce point pour l'instant. Voyons juste un exemple simple :

```
# type couleur = Trefle | Carreau | Coeur | Pique ;;
type couleur = Trefle | Carreau | Coeur | Pique

# [Trefle ; Carreau] ;;
- : couleur list = [Trefle; Carreau]
```

7

Les « définitions »

C'est comme en maths...mais en anglais! Donc *Soit* se dit *Let* :

```
# let x=3*2;;
val x : int = 6
# 2+x;;
- : int = 8
# let pi=acos(-1.);;
val pi : float = 3.14159265358979312
# sin(pi/.2.);;
- : float = 1.
```

Le nom des identificateurs doit commencer par une lettre minuscule.

On peut définir *localement* une variable, c'est-à-dire que sa portée ne dépassera pas l'expression où elle a été définie :

```
# let x = 3 in x + 1 ;;
- : int = 4
```

La définition est bien locale :

```
# x ;;
Characters 0-1:
  x ;;
  ^
Error: Unbound value x
```

```
let (x,y) = (1,2) in x + y ;;
- : int = 3
```

8

Fonctions

On peut créer des fonctions avec **function** :

```
# let delta = function
| (a,b,c) -> b*b-4*a*c;;
val delta : int * int * int -> int = <fun>

# delta(1,1,1);;
- : int = -3
```

Notez le `val delta : int * int * int -> int = <fun>` qui indique que la fonction construite...est une fonction (<fun>) qui va de \mathbb{Z}^3 dans \mathbb{Z} .

On peut de manière plus standard (informatiquement parlant !) définir la fonction par son expression générale :

```
# let discriminant(a,b,c)=b*b-4*a*c;;
val discriminant : int * int * int -> int = <fun>

# discriminant(1,1,1);;
- : int = -3
```

On peut disjoindre des cas : c'est le fameux **filtrage par motif** (« pattern matching »)

```
# let sina = function
| 0. -> 1.
| x -> sin(x)/.x;;
    val sina : float -> float = <fun>

# sina(0.);;
- : float = 1.

# sina(0.1);;
- : float = 0.998334166468281548
```

On peut rendre le paramètre du filtrage explicite :

```
let rec factorielle n = match n with
| 0 -> 1
| n -> n * factorielle (n-1);;
```

On peut travailler avec autre chose que des nombres :

```
# let mystere = function
| (false,true) -> true
| (true,false) -> true
| _ -> false;;
    val mystere : bool * bool -> bool = <fun>
```

Le tiret _ indique « dans tous les autres cas ».

On peut aussi créer des fonctions *polymorphes*, c'est-à-dire qui ne travaillent pas sur des types de variables particuliers.

```
let composee(f,g) = function
| x -> f(g(x));;
    val composee : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

Ici, *f* est toute fonction transformant un type de variable *a* en un type de variable *b* et *g* une fonction transformant un type de variable *c* en un type de variable *a*. La fonction composée transforme bien un type de variable *c* en un type de variable *b*.

Si nous reprenons la fonction définie précédemment :

```
let carre = function
| x -> x*.x;;
    val carre : float -> float = <fun>
```



```
# composee(sina,carre);;
- : float -> float = <fun>

# composee(sina,carre)(3.);;
- : float = 0.04579094280463962
```

9

If...then...else

On peut utiliser une structure conditionnelle pour définir une fonction :

```
let valeur_abs(x)=
  if x>=0 then x
  else -x;;
val valeur_abs : int -> int = <fun>
```

10

Fonctions récursives

C'est une fonction construite à partir d'elle-même. On utilise la même syntaxe que pour une fonction simple mais on fait suivre le **let** d'un **rec** :

```
let rec factorielle = function
| 0 -> 1
| n -> n*factorielle(n-1);;
val factorielle : int -> int = <fun>

let rec fact(n)=
  if n=0 then 1
  else n*fact(n);;
val fact : int -> int = <fun>
```

11

Définition locale de fonctions

```
let nb_sol(a,b,c) =
  let delta(a,b,c) =
    b*b - 4*a*c
  in if delta(a,b,c) = 0 then 1
     else if delta(a,b,c) < 0 then 0
        else 2;;
```

12

Récursion terminale

```
let rec succ = function
| 0 -> 1
| n -> 1 + succ(n-1) ;;
    val succ : int -> int = <fun>

# succ(100000);;
- : int = 100001
# succ(1000000);;
Stack overflow during evaluation (looping recursion?).
```

On arrive à la limite de la pile.

```
let succ(n) =
  let rec local = function
    | (0,acc) -> acc
    | (n,acc) -> local(n-1,1 + acc)
  in local(n,1);;
    val succ : int -> int = <fun>

# succ(100000000);;
- : int = 100000001
```

Cam1 peut dérécurifier les récursions terminales et n'utilise plus sa pile qui ne peut donc plus être saturée.

13

Au cas zoù, on peut programmer en impératif

13 1 Type unit

Il existe des fonctions qui jouent le rôles des procédures dans des langages impératifs et sont à effets de bords. Ocaml étant typé, il y a un type pour ces fonctions : **unit**.

```
# print_string "Tralala" ;;
Tralala- : unit = ()
```

Une autre action extra-fonctionnelle est la modification en place d'un vecteur à l'aide de **<-** :

```
# let v = [|1;2;3|];;
val v : int array = [|1; 2; 3|]
# v.(2) <- 37 ;;
- : unit = ()
# v ;;
- : int array = [|1; 2; 37|]
```

On évitera ce genre d'action extra-fonctionnelle dans la mesure du possible.

13 2 Références

Même si Caml fait une incursion dans la programmation impérative, il la clarifie un peu en obligeant à distinguer le nom d'une référence et son contenu. On ne pourra pas écrire `x := x + 1` comme en C par exemple. La référence est créée avec `ref` et son contenu est accessible en faisant précéder son étiquette d'un `!` :

```
# let x = ref 0 ;;
val x : int ref = {contents = 0}
# x ;;
- : int ref = {contents = 0}
# x + 1 ;;
Characters 0-1:
  x + 1 ;;
  ^
Error: This expression has type int ref
      but an expression was expected of type int
# !x + 1 ;;
- : int = 1
# x := !x + 3 ;;
- : unit = ()
# x ;;
- : int ref = {contents = 3}
# !x ;;
- : int = 3
```

On remarque au passage qu'une réaffectation est bien du type `unit`.

13 3 Boucles

La syntaxe est usuelle mais il faut faire attention à la distinction référence/contenu :

```
(* POUR *)
# let som_ent(n) =
  let s = ref 0 in
  for i=1 to n do
    s := !s + i
  done;
  !s;;
val som_ent : int -> int = <fun>

# som_ent(5) ;;
- : int = 15

(* TANT QUE *)
# let som_ent_w(n) =
  let s = ref 0 in
  let i = ref 0 in
  while (!i <= n) do
    begin
      s := !s + !i;
      incr i;
    end
  end
```

```
    done;  
    !s;;  
val som_ent_w : int -> int = <fun>  
  
# som_ent_w(5) ;;  
- : int = 15
```