CAML pour l'impatient

l'essentiel pour utiliser CAML au lycée

Guillaume CONNAN

IREM de Nantes

3 janvier 2010

Sommaire

Sommaire

Pour tous les O.S. il existe des distributions de CAML *clé-en-main* sur le page de Jean Mouric.
Pour les utilisateurs de Emacs, vous pouvez charger le mode tuareg.

CAML sous Emacs avec le mode tuareg :

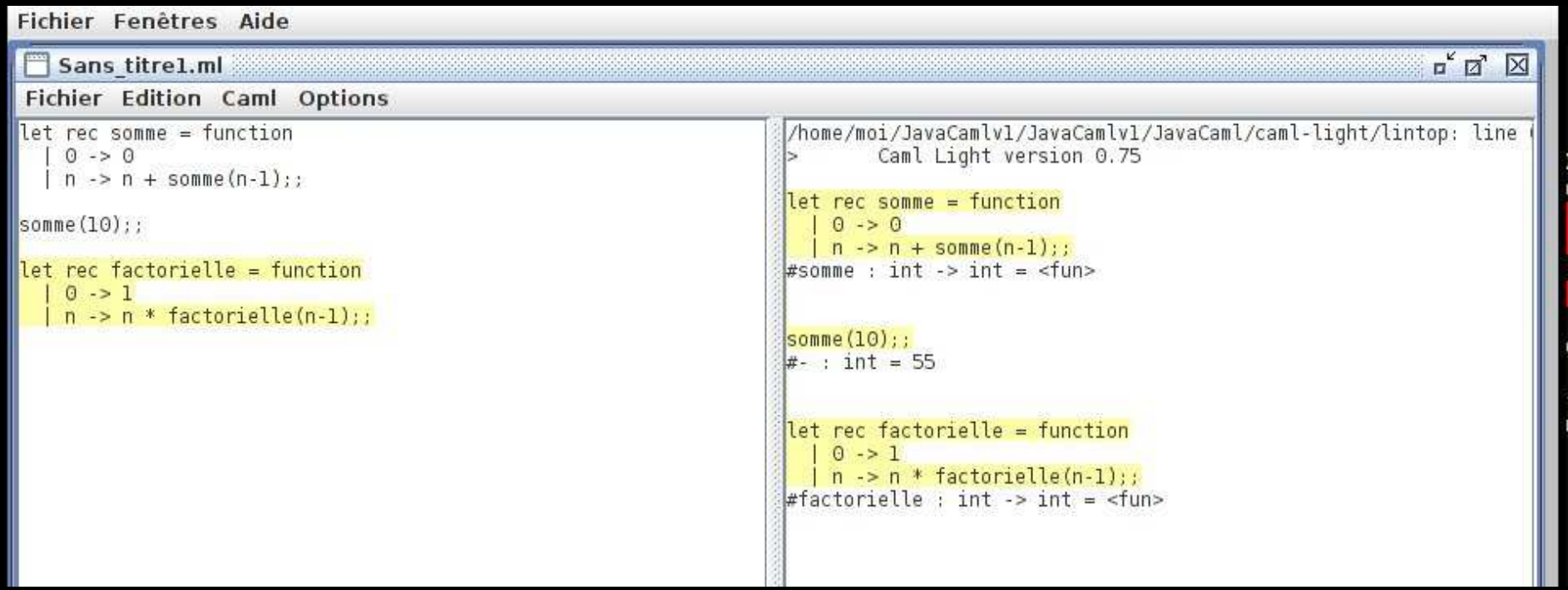
```
File  Edit  Options  Buffers  Tools  Tuareg  Complete  In/Out  Signals  Help
        Objective Caml version 3.11.1


# 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 :



Sommaire

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^{30} et $2^{30} - 1$.

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 $m \times 10^n$. En externe, il apparaît sous forme décimale, le séparateur étant le point.

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

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 $m \times 10^n$. 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 ».

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 = 1.4142135623730951
```

```
# 3.**2.;;
- : float = 9.
# log(2.);;
- : float = 0.69314718055994528 6
# exp(1.);;
- : float = 2.7182818284590451 09
# cos(0.);;
- : float = 1.
# cos(2.*.atan(1.));;
- : float = 6.123031769111886 29e-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
```

Sommaire

Sommaire

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
```

Sommaire

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

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

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

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

Sommaire

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

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

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

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

Sommaire

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.
```

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.
```

attention

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=99 in x+1;;
- : int = 100
# x;;
- : int = 6
# let a=3 and b=2 in a*a+b*b;;
- : int = 13
```

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=99 in x+1;;
- : int = 100
# x;;
- : int = 6
# let a=3 and b=2 in a*a+b*b;;
- : int = 13
```

Sommaire

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 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 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 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 :

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

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

# sina(0.1);;
- : float = 0.99833416646828152
```

On peut disjoindre des cas :

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

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

# sina(0.1);;
- : float = 0.99833416646828152
```

On peut travailler avec autre chose que des nombres :

```
# let mystere = function
  | (true,true) -> true
  | _           -> false;;
    val mystere : bool * bool -> bool = <fun>
# mystere(3>2,0=1-1);;
- : bool = true
# mystere(3>2,0>1);;
- : bool = false
```

remarque

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

On peut travailler avec autre chose que des nombres :

```
# let mystere = function
  | (true,true) -> true
  | _           -> false;;
    val mystere : bool * bool -> bool = <fun>
# mystere(3>2,0=1-1);;
- : bool = true
# mystere(3>2,0>1);;
- : bool = false
```

remarque

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

On peut travailler avec autre chose que des nombres :

```
# let mystere = function
  | (true,true) -> true
  | _           -> false;;
    val mystere : bool * bool -> bool = <fun>
# mystere(3>2,0=1-1);;
- : bool = true
# mystere(3>2,0>1);;
- : bool = false
```

remarque

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.

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.

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
```

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
```

Sommaire

On peut utiliser une structure conditionnelle pour définir une fonction :

```
# let val_abs = function
  | x-> if x>=0 then x else -x;;
  val val_abs : int -> int = <fun>
```

On peut utiliser une structure conditionnelle pour définir une fonction :

```
# let val_abs = function
  | x-> if x>=0 then x else -x;;
  val val_abs : int -> int = <fun>
```

ou bien

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

Sommaire

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>
```