

Deux façons de travailler avec OCaml

Programmation Fonctionnelle

Cours 1

L'interpréteur, expressions simples, définitions

September 18, 2013

1. Travailler avec un compilateur (comme en Java, C, ...) :
 - ┆ permet d'obtenir des exécutables autonomes
 - ┆ nécessite une certaine maîtrise du langage
 - ┆ existe aussi en OCaml (voir le cours 9)
2. Travailler avec un interpréteur interactif :
 - ┆ permet d'expérimenter avec le langage, et d'observer les effets des requêtes une par une
 - ┆ plus adapté pour apprendre un langage

La boucle d'interprétation

Comment lancer l'interpréteur ?

- ▶ L'utilisateur tape une **requête** (on dit encore une **phrase**)
Caml : une **expression** terminée par deux points-virgules consécutifs.
 - ▶ *Caml* analyse la syntaxe, affiche un message d'erreur si cette syntaxe est incorrecte.
 - ▶ Si la syntaxe est correcte, l'interpréteur **infère** (c'est-à-dire calcule) le **type** de l'expression, affiche un message d'erreur si l'expression est mal typée.
 - ▶ Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis affiche le type calculé et la valeur obtenue.
 - ▶ Boucle **Read-Eval-Print** introduite par LISP.
- ▶ Dans une shell : `oc ml`
 - ▶ Mieux : lancer l'interpréteur avec un éditeur de ligne (comme `rlwr p` ou `ledit`) : `ledit oc ml`
 - ▶ Encore mieux : Dans emacs (ou xemacs) : utiliser le mode `tu reg`

Exemples (examples1.ml)

```
(* correct *)  
3*(4+1)-7;;  
  
(* syntax error *)  
17 +;;  
  
(* erreur de typage *)  
42 + "hello ";;  
  
2+3*1;;  
  
5/2;;  
  
-5 mod 3;;
```

Remarques

- ▶ C'est le ; ; qui termine la requête, pas le saut de ligne. Les sauts de ligne à l'intérieur d'une requête sont des espaces comme les autres.
- ▶ Les ; ; ne font pas partie de la syntaxe du langage *Caml* lui-même mais sont spécifiques à l'interpréteur.
- ▶ Commentaires : entre (* et *), éventuellement sur plusieurs lignes.

Inférence de types

Types

- ▶ C'est un des traits principal du langage OCaml
- ▶ L'idée est de combiner la commodité des langages non-typés avec la sûreté des langages typés ! ☺
- ▶ L'ensemble des types OCaml est très riche, on verra certains types en OCaml pendant les semaines à venir.
- ▶ Grande flexibilité due à la **polymorphie** (voir plus tard)
- ▶ Il y a aussi des petites inconvenients qui sont dues à l'inférence de type ☹.
- ▶ Types de base : `int`, `float`, `bool`, ...
- ▶ Types fonctionnels (voir plus tard)
- ▶ Types structurés (voir plus tard)
- ▶ Objets et classes (voir encore plus tard)
- ▶ **il n'y a pas de conversion automatique entre types**

Les entiers

- ▶ Type `int`
- ▶ Opérateurs `+`, `-`, `*`, `/` (division entière !)
- ▶ `mod` pour le reste de la division entière.

Les réels

- ▶ Type `float`
- ▶ Les constantes réelles sont écrites avec un point décimal (5.2), ou en notation scientifique (5e3, 6e-9), ou les deux à la fois.
- ▶ En absence du point et de l'exposant on obtient un entier, ce qui n'est pas compatible avec les réels !
- ▶ Les opérations sur les réels s'écrivent avec un point (`+`, etc.)

Exemples (examples2.ml)

```
(* correct *)  
3.0 +. 0.01;;  
  
3e6 +. 2e5;;  
  
2e-4 *. 0.1;;  
  
(* erreur de typage *)  
2 + 3.0;;  
  
1.0 + 2.0;;  
  
4 *. 56;;
```

Fonctions de conversion

- ▶ Il y des fonctions de conversion dans les deux sens : `float_of_int` et `int_of_float`
- ▶ On applique une fonction à un argument en écrivant le nom de la fonction, suivi par l'argument (éventuellement séparé par des espaces)
- ▶ Il peut être nécessaire de mettre des parenthèses quand l'application de la fonction fait partie d'une expression plus complexe.
- ▶ (plus sur les fonctions plus tard)

Exemples (examples3.ml)

```
float_of_int 17;;

int_of_float 42.3;;

(float_of_int 3) +. 5.8;;

(float_of_int (5 * (int_of_float 2.5))) +. 0.1;;
```

Fonctions sur les réels

- ▶ On a les fonctions habituelles `sqrt`, `sin`, `cos`, `ceil`, `floor`, ...
- ▶ Pour plus d'informations sur les fonctions disponibles, voir le manuel.

Expressions Booléennes

- ▶ Type : `bool`
- ▶ Constantes : `true` et `false`
- ▶ Opérateurs `&&` pour le et, et `||` pour le ou, `not` pour la négation.
- ▶ Attention : `and` n'est pas la conjonction logique !
- ▶ Opérateurs de comparaison (`<`, `>`, `=`, `<=`, `=>`) envoient un `bool` quand appliqués à deux arguments **du même type**

Exemples (examples4.ml)

```
true & true;;

true && true ;;

false || true;;

1 < 7;;

5.0 > "hello";;

(7.56 <= 8e32) && (6 > -3);;
```

Des expressions conditionnelles

- ▶ `if ... then ... else`
- ▶ est une **expression**, pas une instruction !
- ▶ les expressions dans les deux branches doivent avoir le même type (qui est alors le type de l'expression entière)
- ▶ Il y a une valeur par défaut quand la partie **else** manque, on va revenir sur ce point plus tard.

Exemples (examples5.ml)

```
if 1<2 then 6+7 else 67/23;;  
  
if 6=8 then 1 else 77.5;;  
  
(if 6=3+3 then 3<4 else 8 > 7) && 67.8 > 33.1;;  
  
if (if 1=1 then 2=2 else 4.0 > 3.2) then 2<3 else 3
```

Définitions globales d'identificateurs

- ▶ `let nom = expression`
- ▶ Les noms des identificateurs sont formés de lettres, chiffres, `_`, apostrophe `'`, et commencent toujours sur une lettre en minuscule (mais pas les tirets `-`).

Exemples (examples6.ml)

```
let x = 2+3;;  
  
let y = 2*x;;  
  
let x = 42;;  
  
y;;  
  
(* erreur *)  
z;;  
  
let x' = 17;; (* OK *)  
  
let x' = 17;; (* erreur *)  
  
(* ne pas confondre les symboles " , ' et ' *)
```

Définitions locales d'identificateurs

- `let identificateur = exp1 in exp2`
- La **portée** de l'identificateur est *exp2*
- Cette définition peut cacher une définition plus globale d'un autre identificateur avec le même nom.

Exemples (examples7.ml)

```

let x = 4+5 in 2*x;;
x;;
let x = 17;;

let y = x+1 in y / 3;;

let x = 4 in
let y = x+1 in
let x=2*y in x;;

let x = 4 in
( let x = 17 in x+1 ) + x;;

```

Fonctions

- Définition globale d'une fonction à un argument :
`let id1 id2 = exp`
 - Définition locale d'une fonction à un argument :
`let id1 id2 = exp1 in exp2`
 - Règles de portées comme avant
 - Application d'une fonction *f* à un argument *a* :

$$f \ a$$
- Des parenthèses sont nécessaires seulement pour desambiguer dans le contexte d'une expression plus grande.

Exemples (examples8.ml)

```

let f x = x+1;;
f 17;;

let g y = 2*y
in g 42;;

f f 1;;

f (f 1);;

(f f) 1;;

```

Liaison statique

- ▶ une utilisation d'un identificateur global dans la définition d'une fonction fait référence à l'identificateur visible au moment de le définition, pas au moment de l'évaluation de la fonction !
- ▶ dans le cas contraire on parle de liaison dynamique (ce qui n'existe plus dans les langages de programmation modernes).
- ▶ Anglais pour *liaison statique* : **lexical scoping**

Exemples (examples9.ml)

```
let f x = x+1 in
let g y = f (f y) in
let f x = 2*x in
g 5;;
```

```
let f x = x+1;;
let g y = f (f y) ;;
let f x = 2*x;;
g 5;;
```

Les identificateurs ne sont pas de variables !

- ▶ Deux let successives pour des identificateurs avec le même nom : la deuxième cache la première, mais ne change pas la valeur de l'identificateur.
- ▶ La valeur d'un identificateur ne peut être changée !!
- ▶ Il n' a pas de différence fondamentale entre la définition d'un identificateur lié à un entier, et d'un identificateur lié à une fonction (à part du type).
- ▶ (Pourtant, *CamI* permet aussi la programmation impérative. Explication : voir le cours 8 !)

Exemples (examples10.ml)

```
let a = 1;;

let f x = x + a;;

f 2;;

let a = 666;;

f 2;;
```