TP 1 de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

4 février 2015

L'objectif est de présenter les notions élémentaires propres aux langages fonctionnels (calcul d'expressions, expressions fonctionnelles, définition de fonctions, récursivité).

1 Nombres

Le langage Objective Caml connaît les nombres entiers ($[-2^{30}, 2^{30}-1]$) de type **int** et les nombres flottants ($m \times 10^n$) de type **float**. Les opérations usuelles sur les nombres sont les suivantes :

nombres entiers	nombres flottants
`+` addition	`+.` addition
`-` soustraction et moins unaire	`-.` soustraction et moins unaire
`*` multiplication	`*.` multiplication
`/` division entière	`/.` division
`mod` reste de la division entière	

— Écrire une fonction qui calcule $2x^2 + 3x - 2$ sur les entiers.
— Écrire une fonction qui calcule $2x^2 + 3x - 2$ sur les flottants.

2 Expressions conditionnelles

Syntaxe

`if` e_1 `then` e_2 `else` e_3

L'expression e_1 doit être de type *bool*. Ce type contient uniquement deux valeurs, `true` et `false`, appelés *booléens*. Les expressions e_2 et e_3 doivent être du même type.

Contrairement au langage Pascal, la clause `else` est ici obligatoire [1], de façon à ce que la valeur de l'expression soit définie, que la condition soit vraie ou fausse.

— Que retourne l'expression : `if -3 < 0 then 3 else 3` ?
— Écrire une fonction qui calcule la valeur absolue d'un entier.

1. Il existe une forme sans `else`, mais nous déconseillerons pour l'instant son usage.

3 Évaluation d'une expression

Pour calculer la valeur d'une expression, on la tape directement dans la boucle interactive (ou *top-level*), suivie de deux point-virgules. Ainsi l'interprète Caml l'évalue, c'est-à-dire qu'il calcule sa valeur et son type, puis affiche les deux. L'autre approche consiste à utiliser directement le compilateur :

```
ocamlc -c toto.ml
```

compile le fichier `toto.ml` en le fichier objet `toto.cmo`, l'édition de lien est faite par `ocamlc -o toto toto.cmo` si on souhaite nommer `toto` l'exécutable.

4 Liaisons

4.1 Liaisons globales (phrases)

Syntaxe

`let` p = e`;;`

`let` p_1 = e_1 `and` p_2 = e_2`;;`

Après une déclaration de variable globale, l'interprète Caml affiche le type de chaque variable déclarée, ainsi que sa valeur.

Il est important, lors d'une déclaration de variable globale, de ne pas oublier le mot-clé `let` en début d'instruction. En effet p=e est une expression booléenne, qui représente la comparaison de p et de e. La variable p n'étant pas encore définie, cette expression est rejetée par le compilateur.

Quel est l'effet des déclarations globales suivantes :

```
let an = "2003";;
let x = int_of_string(an);;
let nouvel_an = string_of_int(x+1);;
```

4.2 Liaisons locales (expressions)

Syntaxe

`let` p = e_1 `in` e_2

`let` p_1 = e_1 `and` p_2 = e_2 `in` e_3

— Que valent les expressions suivantes :
`let x = 3 in let b = x < 10 in if b then 0 else 10`
`let a = 3.0 and b = 4.0 in sqrt(a*.a+.b*.b)`
où `sqrt` est la racine carrée sur les flottants ?

— Écrire une fonction qui calcule les racines d'un polynôme du second degré qui possède deux racines.

5 Produit cartésien

Syntaxe

On construit un n-uplet par

(e_1, e_2, ... , e_n)

Si les expressions $e_1 \dots e_n$ ont respectivement les types $t_1 \dots t_n$, alors ce n-uplet a le type $t_1 \times \dots \times t_n$. Notons que les t_i ne sont pas forcément identiques.

Exemple :

```
# (3, (if 1 < 2 then 1.0 else 0.1), true);;
- : int * float * bool = (3, 1.0, true)
```

On peut « dé-structurer » un n-uplet e par une construction `let` ou `let in` comme ceci :

`let` $(x_1, x_2, \dots, x_n)$ `=` e`;;`

`let` $(x_1, x_2, \dots, x_n)$ `=` e `in` e'

Exemples :

```
# let paire z = (z + z, z - z);;
val paire : int -> int * int = <fun>
# let (x, y) = paire 3;;
val x : int = 6
val y : int = 0
```

On peut aussi écrire directement une fonction dont l'argument est un n-uplet, comme ceci :

`let` $f\,(x_1, x_2, \dots, x_n)$ `=` e`;;`

Exemple :

```
# let ajoute (x, y) = x + y;;
val ajoute : int * int -> int = <fun>
# ajoute (paire 4);;
- : int = 8
```

6 Récursivité

Toutes les variables d'une expression doivent être définies pour l'évaluation de celle-ci. Cela est vrai pour les définitions de la forme `let` p `=` e, toutes les variables de `e` doivent être connues.

```
# x+2;;
  ^
Unbound value x
# let z x = if x = 0 then 0 else x + z (x-1);;
                                        ^
Unbound value z
# let p = p+1;;
          ^
Unbound value p
```

Pour les définitions récursives de fonctions, il est nécessaire de pouvoir faire appel à la fonction dans sa propre définition. Dans ce cas, la déclaration d'une fonction récursive utilisera la construction `let rec`.

```
# let rec sigma x = if x = 0 then 0 else sigma (x-1) + x;;
val sigma : int -> int = <fun>
# sigma 10;;
- : int = 55
```

7 Expressions fonctionnelles

Jusqu'ici, nous avons utilisé `let` pour définir des fonctions nommées. On peut également définir des fonctions sans leur donner de nom, en utilisant la construction `fun`.

Syntaxe

```
fun x -> e
```

Ceci représente la fonction qui à x associe l'expression e. La variable x peut bien sûr apparaître dans e.

Exemple :

```
# fun x -> x + 3;;
- : int -> int = <fun>
# (fun x -> x + 3) 7;;
- : int = 10
```

Les expressions fonctionnelles sont des valeurs du langage. D'ailleurs, la notation

$$\texttt{let } f\, x = e\texttt{;;}$$

n'est qu'une abréviation pour

$$\texttt{let } f = \texttt{fun } x \texttt{ -> } e\texttt{;;}$$

Une fonction peut être le résultat d'une fonction. Par exemple

```
(fun x -> fun y -> x + y) 3
```

est une fonction à un argument, qui attend un entier et lui ajoute 3. Cette expression est équivalente à

```
fun y -> 3 + y
```

que l'on obtient en remplaçant x par sa valeur dans l'expression de départ.

Une fonction peut être passée en tant que paramètre à une autre fonction. On peut ainsi définir la composition de deux fonctions de la manière suivante :

```
let compose = fun f -> fun g -> fun x -> f (g x)
```

Exemple :

```
let add1 = fun x -> x + 1
let mult5 = fun x -> 5 * x
```

— Donner les valeurs des expressions suivantes :
```
compose add1 mult5 3
compose mult5 add1
```
— Écrire une fonction `deuxfois` qui applique deux fois une fonction `f` à un argument `x`, d'abord sans utiliser `compose`, puis en utilisant `compose`.

8 Portée statique des variables

```
# let p = 10;;
p : int = 10
# let k x = x + p;;
k : int -> int = <fun>
# let p = p+1;;
```

```
p : int = 11
# p;;
- : int = 11
# k 0;;
- : int = 10
```

Comment expliquer les résultats fournis par les commandes ci-dessus ?