I Introduction à OCaml

25 janvier 2024

1. Introduction

Caml est un langage de programmation développé depuis 1985 par l'INRIA. La variante actuellement active de ce langage est OCaml, qui est le langage que nous utiliserons dans le cadre de l'option informatique.

1.1. Paradigmes de programmation

Un *paradigme de programmation* est une façon d'approcher la programmation informatique en traitant d'une certaine manière les solutions aux problèmes. Il existe de multiples paradigmes. Certains langages sont conçus pour supporter un paradigme, d'autres pour en supporter plusieurs, même s'ils en privilégient souvent un en particulier. Voici trois types de paradigmes courants, parmi d'autres :

a. Programmation impérative

Le paradigme originel et le plus courant est celui de la *programmation impérative*. La programmation impérative consiste à donner des ordres à la machine, souvent appelés des "instructions". Ces instructions sont organisées en séquences, et sont exécutées par l'odinateur pour modifier l'état du programme et de la mémoire de la achine, grâce à des affectations successives. Cela nécessite pour le programmeur de connaître en permanence l'état de la mémoire que le programme modifie, ce qui peut se révéler une tâche complexe.

On distingue les instructions des expressions. Par exemple, en Python, `3*x + 1` est une expression, alors que `y = 3*x+ 1` est une instruction.

Le code en résultant est souvent facile à comprendre mais volumineux.

b. Programmation fonctionnelle

La *programmation fonctionnelle* est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Plus précisément, en informatique, on appelle fonction pure une fonction :

— qui renvoie toujours la même valeur pour les mêmes valeurs des arguments (c'est la partie mathématique de la définition) ;
— qui n'utilise ni ne modifie en aucune manière les variables de son environnement (c'est la partie informatique de la définition). On dit qu'elle n'a aucun "effet de bord".

Une fonction pure ne peut utiliser ni ne modifier une variable globale, c'est-à-dire définie en dehors d'elle-même. La programmation fonctionnelle pure consiste à n'écrire que des fonctions pures sans effet de bord et à composer ces fonctions pour construire des programmes plus

complexes. En programmation fonctionnelle (pure), il n'y a donc ni instructions, ni affectations, ni boucles. Elle favorise au contraire l'usage de la récursivité, et le code en résultant est souvent concis.

c. Programmation orientée objet

La *programmation orientée objet* consiste à modéliser un système comme un ensemble d'objets, où chaque objet représente un aspect donné du système. les objets contiennent des fonctions (ou méthodes) et des données. Ils possèdent une structure interne et un comportement, et ils savent interagir avec les autres objets.

d. Python et OCaml

Python est un langage multiparadigme : il favorise les trois types de paradigmes cités précédemment. OCaml est par contre à la base un langage fonctionnel, mais il permet tout de même la programmation impérative ainsi que la programmation orientée objet.

1.2. Typage

En informatique, toute donnée a un *type* : il s'agit de la nature des valeurs qu'elle peut prendre, ainsi que des opérateurs qui peuvent lui être apliqués. En Python, les expressions ont des types qui sont déterminés à l'exécution du programme ; on dit que Python est *dynamiquement typé*.

Le langage Caml est dit fortement typé car : - le typage d'une expression est réalisé au moment de la compilation ; - les conversions implicites de type sont formellement interdites.

Par exemple, en Python, il est possible d'additionner directement un flottant et un entier : l'entier sera dynamiquement converti en flottant au moment de l'exécution. En Caml, cette conversion ne peut pas être implicite, il faut réaliser explicitement la conversion, et utiliser l'opérateur adapté :

[1] :
```
2 + 2.3 ;;
```

```
Line 1, characters 5-8 :
Error : This expression has type float but an expression was expected of type
         int
```

En effet, Caml n'autorise pas la surcharge d'opérateur : le symbole `+` désigne uniquement l'addition des entiers, l'addition des flottants est notée `+.`

De plus en Python une même fonction peut s'appliquer à des objets de type différent. Par exemple, la fonction Python `len` calcule indifféremment la longueur d'une chaîne de caractères ou d'une liste alors qu'en Caml existent deux fonctions distinctes `string_length` et `length`. Python permet également à une fonction de renvoyer des valeurs de types différents (parfois une liste, parfois un entier). Mais OCaml est beaucoup moins permissif et effectue un contrôle strict des types. Ainsi, une fonction ne peut être appliquée qu'à des arguments du même type, et ne renvoie toujours qu'un seul type d'élément.

2. Types de bases

2.1. Unit

Ce type correspond au `None` de Python, c'est à dire le "rien". Il n'existe qu'une seule valeur de ce type, et elle est notée avec des parenthèses :

```
[2] : () ;;
```

```
[2] : - : unit = ()
```

Lorsqu'une fonction ne renvoie aucune valeur, mais se contente de modifier l'environnement (afficher une image, modifier une valeur dans un tableau, imprimer du texte,...), le type renvoyé par cette fonction est le type `unit`.

2.2. Les entiers

Sur les ordinateurs équipés d'un processeur 64 bits, les éléments de type `int` sont les entiers de l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$, les calculs s'effectuant modulo 2^{63} suivant la technique du complément à deux. Il faut donc veiller à ne pas dépasser ces limites sous peine d'obtenir des résultats surprenants :

```
[3] : let a = 2147483648 in a * a ;;
```

```
[3] : - : int = 0
```

```
[4] : let a = 2147483648 in a * a - 1 ;;
```

```
[4] : - : int = -1
```

On remarquera que $2^{31} = 2147483648$. Les valeurs maximales et minimales de type `int` sont accessibles par `max_int` et `min_int` :

```
[5] : max_int ;;
```

```
[5] : - : int = 2147483647
```

```
[6] : min_int ;;
```

```
[6] : - : int = -2147483648
```

Les opérations usuelles sur les entiers se notent `+`, `-`, `*`, `/` (quotient de la division euclidienne), et `mod` (reste de la division euclidienne pour les entiers +). En l'absence de parenthèses, les règles usuelles de priorité s'appliquent.

```
[7] : 5 + 7 mod 2 ;;
```

```
[7] : - : int = 6
```

```
[8] : 3 + 2*4 ;;
```

```
[8] : - : int = 11
```

```
[9] : (-5) mod 2 ;;

[9] : - : int = -1
```

2.3. Les flottants

Le type `float` est celui des nombres à virgule flottante, qui sont des approximations des nombres réels. Les opérateurs sur les flottants se notent `+.`, `-.`, `*.`, `/.` et `**` (élévation à la puissance). Attention, OCaml est fortement typé et on ne peut pas appliquer à un objet de type `int` une fonction s'appliquant à des objets de type `float`.

```
[10] : 0.1 *. 3. ;;

[10] : - : float = 0.300000000000000044
```

Il existe néanmoins des outils de conversion :

```
[11] : int_of_float 5.13 ;;

[11] : - : int = 5

[12] : float_of_int 3 ;;

[12] : - : float = 3.
```

On dispose de plus d'un certain nombre de fonctions mathématiques : `sqrt`, `exp`, `log` (qui désigne le logarithme népérien), `sin`, `cos`, `tan`, `asin`, `acos`, `atan`...

```
[13] : 3.**2. +. sqrt(4.) ;;

[13] : - : float = 11.

[14] : sin(asin(0.3)) ;;

[14] : - : float = 0.3
```

a. Exercice 1

Que se passe-t-il si on écrit la phrase suivante :

```
[ ] : tan ;;

[ ] : tan(1.) +. tan(-1.) ;;
```

2.4. Les booléens

Le type `bool` comporte deux constantes : `true` et `false` (sans majuscule) et dispose des opérateurs logiques *non* (`not`), *et* (`&&`), *ou* (`||`).

Les opérateurs `&&` et `||` fonctionnent suivants le principe de l'évaluation paresseuse : `p && q` ne va évaluer `q` que si `p` est vraie, et `p || q` ne va évaluer `q` que si `p` est fausse.

```
[17] : not  ; ;
```

```
[17] : -  : bool -> bool = <fun>
```

```
[18] : not (1 = 2) || (1 / 0 = 1)  ; ;
```

```
[18] : -  : bool = true
```

```
[19] : (3 > 0) && (5/0 = 1)  ; ;
```

```
[19] : Exception : Division_by_zero.
```

Le test d'égalité se fait par un simple `=` au lieu de `==` en Python et le test de différence se fait par `<>`. Si le `or` est synonyme du `||`, on prendra garde à ne pas utiliser le `and` qui a une autre signification en OCaml.

2.5. Les caractères et chaînes de caractères

Caml distingue les caractères, de type `char`, et les chaînes de caractères, de type `string`. Les caractères sont entourés de guillemets simples, les chaînes de caractères de guillemets doubles. Les chaînes de caractères disposent d'un opérateur de concaténation, noté `^`

```
[20] : 'a'  ; ;
```

```
[20] : -  : char = 'a'
```

```
[21] : "a"  ; ;
```

```
[21] : -  : string = "a"
```

```
[22] : "MPSI"  ; ;
```

```
[22] : -  : string = "MPSI"
```

```
[23] : "MP"^"SI"  ; ;
```

```
[23] : -  : string = "MPSI"
```

La fonction `length` du module `String` permet d'obtenir la longueur d'une chaîne de caractères.

```
[24] :  String.length "anticonstitutionnellement"  ; ;
```

```
[24] : -  : int = 25
```

la longueur d'une chaîne est directement liée au nombre d'octets utilisés pour représenter un caractère. Attention avec les caractères accentués (qui ne sont pas ASCII) !

```
[25] : String.length "à"  ; ;
```

```
[25] : -  : int = 2
```

Il est possible d'obtenir un caractère d'indice donné :

```
[26] : "anticonstitutionnellement".[0] ;;
```

```
[26] : - : char = 'a'
```

2.6. Les tuples

Les tuples (n-uplets en français) sont similaires aux tuples de Python : ils permettent de construire des séquences de valeurs de type possiblement différents. Le type d'un tuple de la forme `(t1 , t2 , . . . , tp )` est le produit cartésien des types de ses éléments, par exemple `bool * int * float` dans l'exemple ci-dessus. Comme on le voit, les parenthèses sont facultatives. On peut récupérer dans des variables les composantes d'un tuple :

```
[27] : let u = "MPSI", 2024 ;;
       let a, b = "MPSI", 2024 ;;
```

```
[27] : val u : string * int = ("MPSI", 2024)
       val a : string = "MPSI"
       val b : int = 2024
```

3. Définitions globales et locales

3.1. Définition globale

Pour déclarer une variable, on utilise `let`. La déclaration permet de déclarer une variable globale qui ne change pas à moins qu'une autre déclaration globale ne l'écrase. Parler de variable ici est donc abusif : on parle plutôt une *constante*, c'est-à-dire de valeur non modifiable. En effet, on effectue simplement une liaison entre un nom (le nom de la variable) et une valeur.

```
[28] : let n = 2 + 3 ;;
       let n = n + 1 ;;
       let n = 1 ;;
```

```
[28] : val n : int = 5
       val n : int = 6
       val n : int = 1
```

3.2. Définition locale

Lorsqu'il lit une phrase `let` x `=` e `;;` où x est un nom de variable et e une expression, Caml évalue l'expression e et ajoute dans l'environnement des variables l'association entre x et la valeur de e.

La syntaxe `let` x `=` e `in` permet de définir temporairement un nom uniquement pour le calcul courant. Si la variable utilisée était déjà associée à une valeur, celle-ci est temporairement oubliée, mais retrouvée à la fin de l'exécution.

```
[29] : let a = 5 ;;
       a ;;
       let a = 3 in a + 1 ;;
```

```
[29] : val a  : int = 5
       -  : int = 5
       -  : int = 4
```

Si le nom de la variable n'était pas lié à une valeur avant la déclaration locale, il ne l'est toujours pas après :

```
[30] : let y = 5.0 in y**2. ;;
       y ;;

Line 2, characters 0-1 :
Error : Unbound value y

[30] : -  : float = 25.
```

3.3. Définitions simultanées

Le mot-clé `and` permet les définitions multiples, mais les valeurs ne deviennent visibles qu'après toutes les déclarations simultanées.

```
[31] : let a = 3 and b = 5 ;;

[31] : val a  : int = 3
       val b  : int = 5

[32] : let c = 2 and d = c + 1 ;;

Line 1, characters 18-19 :
Error : Unbound value c

[33] : let c = 2 ;;
       let d = c + 1 ;;

[33] : val c  : int = 2
       val d  : int = 3
```

4. Expressions conditionnelles

Une expression conditionnelle est une expression de la forme `if` e_1 `then` e_2 `else` e_3, où e_1 est une expression booléenne. Si la valeur de e_1 est `true`, l'expression e_2 est évaluée et sa valeur est retournée ; sinon, ce sera la valeur de e_3 qui sera retournée.

```
[34] : if true then 1 else 0 ;;

[34] : -  : int = 1
```

OCaml étant fortement typé, e_1 et e_2 doivent être de même type, sinon l'interpréteur avertit d'une erreur :

```
[35] : if true then 1.0 else 0 ;;
```

```
Line 1, characters 22-23 :
Error : This expression has type int but an expression was expected of type
          float
  Hint : Did you mean `0.' ?
```

Le résultat, qui est du même type que e_1 et e_2, peut être réutilisé dans une autre expression :

[36] :
```
let x = 2 and y = 3 ;;
let maxi = if x < y then y else x ;;
```

[36] :
```
val x : int = 2
val y : int = 3
val maxi : int = 3
```

Le `else` et le deuxième bloc d'instructions sont facultatifs, mais il est alors nécessaire que le premier bloc d'instruction soit de type `unit` :

[37] :
```
if true then print_string "c'est vrai !" ;;
```

```
c'est vrai !
```

[37] :
```
- : unit = ()
```

Si un bloc d'instruction contient plus d'une instruction, il est nécessaire d'utiliser un parenthésage. Ce dernier peut se faire avec des parenthèses, ou avec les mots clés `begin...end` :

[38] :
```
let x = if 2 > 3 then 5 else begin print_int 3 ; 3 end ;;
```

```
3
```

[38] :
```
val x : int = 3
```

a. Exercice 2

[39] :
```
x, y ;;
```

[39] :
```
- : int * int = (3, 3)
```

L'expression suivante est-elle acceptée par Caml ?

[ ] :
```
let z = 3 + (if x < y then y else x) ;; (* x et y sont des entiers *)
```

5. Fonctions

5.1. Expressions fonctionnelles

En Caml, les fonctions ont un statut de première classe, c'est-à-dire qu'elles ont le même statut que les autres objets. Pour construire une fonction nommée, on procède comme avec n'importe quelle autre objet, en utilisant `let`.

Pour appliquer une expression fonctionnelle f à un argument e, on écrit tout simplement $f\,e$.

Les valeurs fonctionnelles à une variable sont de la forme `fun` v `->` e, où v est un nom de variable et e une expression.

Il existe deux syntaxes possibles pour créer une fonction d'une variable :

[41] :
```
let f = fun x -> x*x  ; ;
f 4 ; ;
```

[41] :
```
val f  : int -> int = <fun>
-  : int = 16
```

Ou :

[42] :
```
let f x = x*x  ; ;
f 4 ; ;
```

[42] :
```
val f  : int -> int = <fun>
-  : int = 16
```

On remarque que Caml devine tout seul le type de la fonction sans l'exécuter (et sans qu'on lui indique le type de son argument).

Par ailleurs, l'application d'une fonction est prioritaire : `f 4 + 2` est équivalent à `(f 4) + 2`.

[43] :
```
 f (4+2)  ; ;
 f 4 + 2 ; ;
```

[43] :
```
-  : int = 36
-  : int = 18
```

Le mot-clé `fun` permet aussi d'utiliser une fonction sans la nommer.

[44] :
```
(fun x -> x^x) "to" ; ;
```

[44] :
```
-  : string = "toto"
```

On peut également passer une fonction en argument, comme n'importe que objet.

[45] :
```
let h f = f 4 ; ;
h (fun x -> 2*x) ; ;
```

[45] :
```
val h  : (int -> 'a) -> 'a = <fun>
-  : int = 8
```

5.2. Fonctions à plusieurs variables

La manière naturelle de construire une fonction à plusieurs variables en Caml est d'utiliser l'expression `fun` v_1 v_2 ... v_n `->` e, où v_1,...,v_n sont des noms de variables et e une expression. On dispose d'une autre syntaxe possible dans le cas d'une fonction nommée.

[46] :
```
fun x y -> x / y ; ;
```

[46] :
```
-  : int -> int -> int = <fun>
```

```
[47] : let p = fun x y -> x*y ; ;

[47] : val p  : int -> int -> int = <fun>

[48] : let p x y = x*y ; ;

[48] : val p  : int -> int -> int = <fun>
```

Il y a en fait un autre moyen d'écrire une fonction à plusieurs variables, en utilisant un produit cartésien :

```
[49] : let somme1 = fun x y -> x + y ; ;

[49] : val somme1  : int -> int -> int = <fun>

[50] : let somme2 = fun (x, y) -> x + y ; ;

[50] : val somme2  : int * int -> int = <fun>
```

On remarque que ces deux fonctions ont des types différents. - La fonction `somme2` est en fait une fonction à une seule variable, de type composé. - La fonction `somme1` est considérée comme la cascade $x \mapsto (y \mapsto (x+y))$. Cette façon d'écrire la fonction est appelée la version *curryfiée*. Il est alors possible de définir des fonctions partielles :

```
[51] : let incremente = somme1 1 ; ;

[51] : val incremente  : int -> int = <fun>

[52] : incremente 3 ; ;

[52] : -  : int = 4
```

Sauf très bonne raison de faire autrement, on préférera systématiquement la version curryfiée.

Un dernier exemple, avec d'autres types que `int` :

```
[53] : let f x y = if (y>1)||x then 2.2 else 5.1  ; ;

[53] : val f  : bool -> int -> float = <fun>
```

5.3. Fonctions récursives

Considérons la fonction `fact` définie ci-dessous :

```
[ ] : let rec fact n =
        if n = 0
        then 1
        else n * fact (n-1)
         ; ;
```

Le mot-clé `rec` indique que nous avons défini un objet *récursif*, c'est-à-dire un objet dont le nom intervient dans sa propre définition. Nous approfondirons ultérieurement la notion de fonctions récursives (notamment en termes de terminaison et de complexité).

a. Exercice 3

1. Quel est le type de la fonction `fact` ?
2. Pour quelles valeurs de `n` la fonction termine-t-elle ? Quelle est le nombre de multiplications effectuées dans ce cas ?
3. Comment expliquer le résultat suivant ?

[55] :
```
fact 64 ; ;
```

[55] :
```
-  : int = 0
```

Il est aussi possible de définir deux fonctions mutuellement récursives à l'aide du mot-clé `and` :

[56] :
```
let rec u n =
  if n = 0 then 1 else 3*u(n-1) + 2*v(n-1)
and v n =
  if n = 0 then 2 else 2*u(n-1) + 3*v(n-1)
 ; ;
```

[56] :
```
val u  : int -> int = <fun>
val v  : int -> int = <fun>
```

5.4. Polymorphisme

On a constaté que Caml est capable de reconnaître automatiquement le type d'une fonction. Par exemple, pour la fonction `fun x y -> x + y`, la présence de l'opérateur `+` permet d'associer à cette fonction le type `int -> int -> int`.

Il peut en revanche arriver qu'une fonction puisse s'appliquer indifféremment à tous les types ; on dit alors que la fonction est *polymorphe*. Caml utilise alors les symboles `'a`, `'b`, `'c`,... pour désigner des types quelconques.

[57] :
```
let premier = fun (x, y) -> x ; ;
```

[57] :
```
val premier  : 'a * 'b -> 'a = <fun>
```

[58] :
```
let compose f g = fun x -> f (g x) ; ;
```

[58] :
```
val compose  : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Nous avons vu que les opérations algébriques, l'addition par exemple, ne sont pas polymorphes : l'opérateur infixe `+` est de type `int -> int` alors que `+.` est de type `float -> float`. En revanche, les opérations de comparaison telle l'égalité (notée `=`) ou les inégalités (`<`, `>`, `<=`, `>=`) le sont. On peut utiliser le même opérateur pour comparer entre eux des entiers, des nombres complexes ou des chaînes de caractères (et plus généralement tout objet pour lesquels la comparaison a un sens), car c'est leur représentation machine qui est comparée.

[59] :
```
1>=2 ; ;
1.>=2. ; ;
```

[59] :
```
-  : bool = false
-  : bool = false
```

6. Filtrage

Dans les deux exemples de fonctions récursives, on distingue deux cas, pour lesquels on calcule de manières différente la valeur à renvoyer. Ces deux cas portent sur la valeur de `n`. Il est alors possible d'utiliser une construction très puissante, et omnprésente en OCaml, nommée *filtrage*.

[60] :
```
let rec fact n =
  match n with
  | 0 -> 1
  | _ ->n*(fact (n-1))
 ;;
```

[60] :
```
val fact  : int -> int = <fun>
```

Chaque cas de filtrage est formé d'un *motif*, suivi d'une flèche, suivi d'une expression. On a ici deux cas de filtrage. On considère leurs motifs un par un, jusqu'à en trouver un qui *filtre* la valeur n. On exécute alors l'expression à droite de la flèche correspondante et on renvoie sa valeur.

Le premier cas de filtrage a pour motif `0`. Si `n` vaut 0, on dit que ce motif filtre n, auquel cas on renvoie `1.

Le troisième cas a pour motif `_`. Ce motif filtre toute valeur. Si n n'a été pas été filtré par un des motifs précédents, il est filtré par celui-ci. On exécute alors l'expression à droite de la flèche, et on renvoie la valeur ainsi calculée.

Lorsque le motif est un nom, il s'accorde avec n'importe quelle valeur, et la reçoit dans l'expression exécutée. Par exemple, la définition de la fonction sinus cardinal sera :

[61] :
```
let sinc x =
    match x with
    | 0. -> 1.
    | x -> sin (x) /. x  ;;
```

[61] :
```
val sinc  : float -> float = <fun>
```

Le motif 0. est une constante qui ne concorde qu'avec la valeur 0. ; le motif x est une variable qui concorde avec n'importe quel nombre flottant.

a. Exercice 4

Comment interpréter l'avertissement et le résultat de l'appel dans les lignes suivantes ?

[62] :
```
let eq x y =
  match y with
  |x -> true
  |_ -> false ;;

eq 2 3 ;;
```

```
Line 4, characters 3-4 :
Warning 11 [redundant-case] : this match case is unused.
```

```
[62] : val eq  : 'a -> 'b -> bool = <fun>
       -  : bool = true
```

Pour corriger ce problème, il est possible d'utiliser un *motif gardé* :

```
[63] : let eq x y =
         match y with
         |y when x=y -> true
         |_ -> false ; ;

       eq 2 3 ; ;
```

```
[63] : val eq  : 'a -> 'a -> bool = <fun>
       -  : bool = false
```

Il aurait bien sûr été plus simple d'écrire cette fonctions de la manière suivante :

```
[64] : let eq x y = x = y ; ;
       eq 2 3 ; ;
```

```
[64] : val eq  : 'a -> 'a -> bool = <fun>
       -  : bool = false
```

Remarquons que la fonction `sinc` aurait aussi pu être écrite ainsi :

```
[65] : let sinc x =
           match x with
           | 0. -> 1.
           | _ -> sin (x) /. x
       ; ;
```

```
[65] : val sinc  : float -> float = <fun>
```

Attention, l'ordre dans lequel on essaye de faire correspondre un motif et une valeur a de l'importance. Par exemple, la définition ci-dessous est incorrecte, car la valeur 0. s'accorde avec le premier motif :

```
[66] : let sinc x =
           match x with
           | x -> sin (x) /. x
           | 0. -> 1.
       ; ;

       sinc 0. ; ; ;
```

```
Line 4, characters 6-8 :
Warning 11 [redundant-case] : this match case is unused.
Line 7, characters 11-12 :
Error : Syntax error
```

```
[66] : val sinc  : float -> float = <fun>
       -  : float = nan
```

Remarquez que l'interpréteur indique lorsqu'un cas est inutile. Il indique également les filtrages non exhaustifs, comme dans l'exemple suivant simulant le lancer d'un dé à 4 faces :

[67] :
```
let de n =
    match n with
    | 1 -> "tourner à gauche"
    | 2 -> "tourner à droite"
    | 3 -> "continuer tout droit"
    | 4 -> "faire demi-tour"
;;
```

```
Lines 2-6, characters 4-28 :
Warning 8 [partial-match] : this pattern-matching is not exhaustive.
Here is an example of a case that is not matched :
0
```

[67] :
```
val de  : int -> string = <fun>
```

Un dernier exemple d'utilisation du symbole _ :

[68] :
```
let k x y =
    match x,y with
    | 0,_ -> 1
    | _,0 -> 1
    | _ -> 0
;;
```

[68] :
```
val k  : int -> int -> int = <fun>
```

7. Exercices divers

7.1. Exercice 5

Prévoir les réponses de l'interprète de commandes après la suite de définitions suivantes :

[ ] :
```
let a = 1 ;;
```

[ ] :
```
let f x = a * x ;;
```

[ ] :
```
let a = 2 in f 1 ;;
```

[ ] :
```
let a = 3 and f x = a * x ;;
```

[ ] :
```
f 1 ;;
```

7.2. Exercice 6

Donner une expression Caml dont le type est :

— `int -> int -> int`

— `(int * int) -> int`

— `int -> (int * int)`

— `(int -> int) -> int`

— `int -> (int -> int)`

— `int -> (int -> int) -> int`

7.3. Exercice 7

Quelle est le type des expressions suivantes ?

— `fun x y z -> x y z ; ;`
— `fun x y z -> x (y z) ; ;`
— `fun x y z -> (x y) + (x z) ; ;`

7.4. Exercice 8

— Écrire la fonction `curry` qui prend en argument une fonction ayant deux paramètres parenthésés et la transforme en une fonction non parenthésée. Quelle est son type ?

— Écrire la fonction réciproque `uncurry`. Quelle est son type ?

7.5. Exercice 9

Donner trois fonctions Caml calculant les coefficients du binôme $\binom{n}{p}$ à l'aide des méthodes suivantes :

— avec la formule : $\binom{n}{p} = \frac{n!}{p!(n-p)!}$

— avec la relation : $\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$

— avec la relation : $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$

7.6. Exercice 10

Un nombre de Hamming est un entier qui ne comporte que des 2, 3 et 5 dans sa décomposition en facteurs premiers. Ecrire une fonction hamming de type int->bool qui détermine si un entier est un entier de Hamming.

7.7. Exercice 11

Coder en OCaml les deux fonctions suivantes : $f : (p,q) \mapsto \begin{cases} q & \text{si } p = 0 \\ f(p-1,q)+1 & \text{sinon} \end{cases}$ et g : $(p,q) \mapsto \begin{cases} q & \text{si } p = 0 \\ g(p-1,q+1) & \text{sinon} \end{cases}$. Que calculent-elles ?