INTRODUCTION AU LANGAGE OCAML

1 À propos d'OCaml

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 Un langage fonctionnel

Caml se range principalement dans la catégorie des langages *fonctionnels*, même s'il permet aussi une programmation impérative, et même orientée objet dans le cas d'OCaml.

Dans un langage impératif, on utilise des suites d'instructions pour modifier l'état de la mémoire. On distingue alors les instructions des expressions. Par exemple, en Python, `3*x + 1` est une expression, alors que `y = 3*x + 1` est une instruction.

Dans un langage fonctionnel tel que Caml, on met en avant la notion d'expression et l'application de fonctions. En OCaml, la notion d'instruction n'existe pas vraiment : on ne travaille qu'avec des expressions.

1.2 Un langage fortement typé

Tout objet défini par un langage de programmation possède un *type* (`int`, `float`, etc.).

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 :

```
2 + 2.3
```

```
      File "[12]", line 1, characters 5-8:
  1 |  2 + 2.3
           ^^^
  Error: This expression has type float but an expression was expected of type
           int
```

```
float_of_int(2) +. 2.3
```

```
- : float = 4.3
```

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

2 Types de bases

2.1 Les entiers

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

```
5 + 7 mod 2
```

```
- : int = 6
```

```
3 + 2*4
```

```
- : int = 11
```

```
5/2*2
```

```
- : int = 4
```

Sur une architecture 64 bits, les éléments de type `int` sont les entiers de l'intervalle $[\![-2^{62}, 2^{62}-1]\!]$, codés en complément à deux sur 63 bits. Il faut donc prendre soin à ne pas dépasser ces limites, sous peine d'obtenir des résultats différents de ceux attendus.

```
2147483648*2147483648
```

```
- : int = -4611686018427387904
```

```
2147483648*2147483648 - 1
```

```
- : int = 4611686018427387903
```

Les valeurs maximales et minimales de type `int` sont accessibles par `max_int` et `min_int`

```
max_int
```

```
- : int = 4611686018427387903
```

```
min_int
```

```
- : int = -4611686018427387904
```

2.2 Les flottants

Les opérateurs sur les flottants se notent `+.`, `-.`, `*.`, `/.` et `**` (élévation à la puissance).

Les calculs effectués sur les flottants sont bien souvent approchés.

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`...

```
3.**2. +. sqrt(4.)
```

```
- : float = 11.
```

```
sin(asin(0.5))
```

```
- : float = 0.5
```

Exercice

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

```
tan(1) +. tan(-1)
```

2.3 Les booléens

Le type `bool` comporte deux constantes : `true` et `false` 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.

```
(2. < 3.) || (1/0 = 1);;
```

```
- : bool = true
```

```
not (2. < 3.) || (1/0 = 1);;
```

```
    Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from Toploop.load_lambda in file "toplevel/toploop.ml", line 212, characters 17-27
```

```
(3 < 0) && (5/0 = 1);;
```

```
- : bool = false
```

```
(3 > 0) && (5/0 = 1);;
```

```
    Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from Toploop.load_lambda in file "toplevel/toploop.ml", line 212, characters 17-27
```

2.4 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 ^

```
'a'
```

```
- : char = 'a'
```

```
"a"
```

```
- : string = "a"
```

```
"MPSI"
```

```
- : string = "MPSI"
```

```
"MP"^"SI"
```

```
- : string = "MPSI"
```

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

```
String.length "anticonstitutionnellement";;
```

```
- : int = 25
```

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

```
"anticonstitutionnellement".[5];;
```

```
- : char = 'o'
```

2.5 Les tuples

Il est possible de définir le produit cartésien de deux ou plusieurs types : si `x` est une variable de type `'a` et si `y` est une variable de type `'b`, alors `(x,y)` est une variable de type `'a*'b`.

```
"MPSI", 2022
```

```
- : string * int = ("MPSI", 2022)
```

3 Définitions globales et locales

En Caml, le mot-clé `let` permet d'attribuer un nom à une valeur.

```
let n = 2 + 3;;
```

```
val n : int = 5
```

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.

```
let a = 5;;
```

```
val a : int = 5
```

```
let a = 3 in a + 1;;
```

```
- : int = 4
```

```
a;;
```

```
- : int = 5
```

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

```
let a = 3 and b = 5;;
```

```
val a : int = 3
val b : int = 5
```

```
let c = 2 and d = c + 1 ;;
```

```
    File "[43]", line 1, characters 18-19:
1 | let c = 2 and d = c + 1 ;;
                      ^
Error: Unbound value c
```

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.

```
let x = 2 and y = 3;;
```

```
val x : int = 2
val y : int = 3
```

```
let maxi = if x < y then y else x;;
```

```
val maxi : int = 3
```

Exercice

Que se passe-t-il lorsqu'on utilise l'expression conditionnelle suivante ?

```
if x > y then 3 else 1.2;;
```

Exercice

L'expression suivante est-elle acceptée par Caml ?

```
let z = 3 + (if x < y then y else x);;
```

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.

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

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 $f\ e$.

```
fun x -> x + x;;
```

```
- : int -> int = <fun>
```

```
let f = fun x -> x*x ;;

f 4;;
f 4 + 2;;

(fun x -> x^x) "to";;
```

```
val f : int -> int = <fun>

- : int = 16

- : int = 18

- : string = "toto"
```

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`.

Dans le cas des fonctions nommées, on dispose d'une syntaxe alternative :

```
let g x = x + 1;;
```

```
val g : int -> int = <fun>
```

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.

```
fun x y -> x + y;;
```

```
- : int -> int -> int = <fun>
```

```
let p = fun x y -> x*y;;
```

```
val p : int -> int -> int = <fun>
```

```
let concat x y = x^y;;
```

```
val concat : string -> string -> string = <fun>
```

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

```
let somme1 = fun x y -> x + y;;
```

```
val somme1 : int -> int -> int = <fun>
```

```
let somme2 = fun (x, y) -> x + y;;
```

```
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 :

```
let incremente = somme1 1;;
```

```
val incremente : int -> int = <fun>
```

```
incremente 3;;
```

```
- : int = 4
```

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

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)
;;
```

```
val fact : int -> int = <fun>
```

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).

Exercice

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 ?

```
fact 64;;
```

```
- : int = 0
```

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

```
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)
;;
```

```
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.

```
let premier = fun (x, y) -> x;;
```

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

6 Filtrage

Dans les fonctions précédentes, 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 nommée *filtrage*.

Pour tester la parité d'un entier naturel, on pourrait par exemple écrire la fonction `est_pair` suivante (très inefficace...)

```
let rec est_pair n =
  match n with
  | 0 -> true
  | 1 -> false
  | _ -> est_pair (n-2)
;;
```

```
val est_pair : int -> bool = <fun>
```

Chaque cas de filtrage est formé d'un *motif*, suivi d'une flèche, suivi d'une expression. On a ici trois 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 `true`.

Le second cas a pour motif `1`. Si `n` vaut 1, on dit que ce motif filtre n, auquel cas on renvoie `false`.

Le troisième cas a pour motif `_`. Ce motif filtre toute valeur. Si n n'a été pas été filtré par un des deux 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.

Exercice

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

```
let eq x y =
  match y with
  |x -> true
  |_ -> false;;
```

```
File "[71]", line 4, characters 3-4:
4 |   |_ -> false;;
       ^
Warning 11: this match case is unused.
```

```
val eq : 'a -> 'b -> bool = <fun>
```

```
eq 2 3;;
```

```
- : bool = true
```

Exercices divers

Exercice 1

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;;
```

Exercice 2

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`

Exercice 3

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);;`

Exercice 4

- 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 ?

Exercice 5

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 : $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$