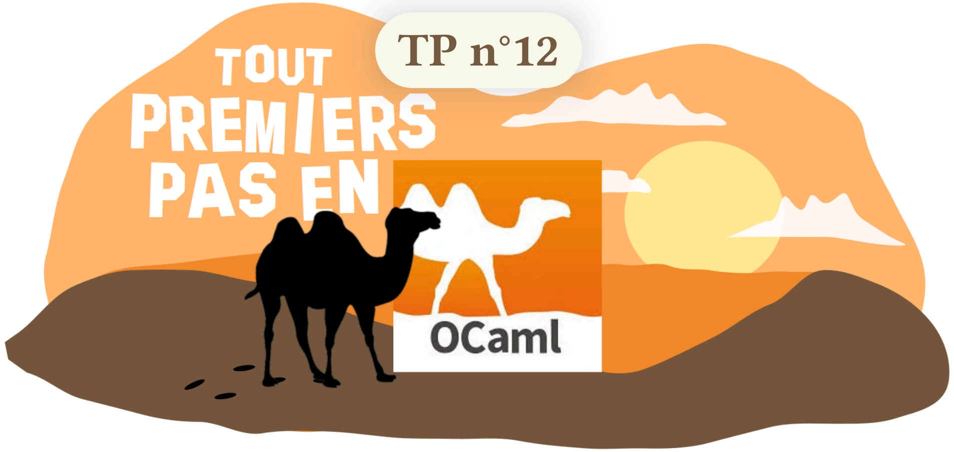



# I. OCaml : un langage interpreté

## Déclaration globale

Une **déclaration globale** permet d'associer un **identifiant** (un nom) à une **valeur** obtenue en **évaluant** une **expression**. On parle aussi de **liaison globale**. On utilise la syntaxe `let <id> = <expr>` avec `<id>` l'identifiant choisi (qui doit commencer par une lettre minuscule) et `<expr>` l'expression à évaluer avant de la lier.

## L'interpréteur OCaml

`ocaml`

Dans un terminal, taper la commande **ocaml**. Cela ouvre un **interpréteur** du langage OCaml.

Un croisillon (#) vous invite à écrire une déclaration globale, qu'il faudra explicitement clore avec deux points-virgules pour indiquer à l'interpréteur qu'il s'agit de la fin de ce qui doit être évalué. L'interpréteur va alors évaluer l'expression donnée et la lier à l'identifiant, avant de vous rendre la main à nouveau pour écrire une nouvelle déclaration globale.

**1** Ouvrir cet interpréteur OCaml dans un terminal. Écrire chacune des déclarations suivantes, en prenant soin de rajouter les deux points virgules et d'appuyer sur **Entrée** à chaque fois.

`let a = 108` `let b = 3.14` `let c = 'S'` `let d = true` `let e = "civ"`

**2** Qu'affiche l'interpréteur, et pourquoi ? Noter les types obtenus : ces cinq types forment les types simples en OCaml. Il en existe un sixième qui sera présenté au prochain TP.



# II. Le type int

Le type `int` permet de représenter des entiers relatifs (positifs ou négatifs). Le langage OCaml met à disposition les opérateurs arithmétiques usuels `+`, `-`, `*`, `/` pour former des expressions entières. On prendra soin de placer des espaces autour des opérateurs binaires. Attention, la puissance entière (notée `**` en Python) n'existe pas en OCaml.

**3** Prédire précisément ce que va afficher l'interpréteur concernant les déclarations globales suivantes. Il est possible de réutiliser des identifiants précédemment définis dans de nouvelles expressions.

`let v1 = 42 + 66` `let v2 = v1 - 10 * 2` `let q = 10 / 3` `let m = v2 / q * 2`

**4** Écrire et évaluer ces déclarations globales dans l'interpréteur. Vérifier que vos prédictions sont correctes. Quelles sont les priorités entre les opérateurs ?

**5** Il existe également un opérateur modulo. Celui-ci est noté `mod`, et non `%` comme en Python ou en C. Aussi, son comportement est légèrement différent. Tester dans l'interpréteur :

`let m1 = 10 mod 3` `let m2 = 10 mod 0` `let m3 = -10 mod 3` `let m4 = 10 mod -3`

On retiendra que si `a mod b` est non nul, alors il est du même signe que `a`. Cela est parfois une source de bug.

**6** En OCaml, on peut mettre des **underscores** _ pour améliorer la lisibilité d'un entier écrit en clair. Vérifier dans l'interpréteur ce qu'il se passe quand on cherche à réaliser la déclaration globale `let grand = 10_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000`.

**7** La valeur du plus grand entier représentable en OCaml est liée à l'identifiant `max_int`. De même, on peut utiliser `min_int` si l'on souhaite la valeur du plus petit entier. À l'aide de l'interpréteur, trouver sur combien de bits sont encodés les entiers en OCaml. Que vaut l'expression `max_int + 1` ?

Toutes ces liaisons globales restent présentes jusqu'à la fin de l'exécution, ou ici tant que l'interpréteur est ouvert. On a vu qu'il est possible de réutiliser les identifiants précédents dans de nouvelles expressions. On peut également uniquement écrire un identifiant (au lieu d'une déclaration globale) pour retrouver sa valeur.

**8** Dans l'interpréteur, écrire les identifiants suivants puis ajouter les deux points-virgules et appuyer sur **Entrée**. Quel affichage obtient-on ? Pourquoi cela ?

`a` `e` `let e = 108` `e`

## III. Le type float

Le type `float` permet de représenter des nombres flottants qui suivent la norme IEEE754, comme en C. Ces valeurs sont encodées sur 64 bits en double précision. OCaml détecte automatiquement qu'un nombre est un flottant s'il contient un point, séparant la partie entière de la partie fractionnaire.

**9** Prédire ce que va afficher l'interpréteur pour ces déclarations globales, puis vérifier en pratique :

```
let f1 = 108.
```

```
let f2 = 4.1
```

```
let f3 = 12
```

```
let f4 = 2.0 + 4.
```

Ce dernier exemple est un piège classique. Contrairement au langage C ou au langage Python, les opérateurs arithmétiques d'OCaml sont différents pour les flottants, et doivent contenir un point aussi. Tout comme le C, OCaml est un langage **fortement typé**, dans lequel chaque valeur possède un type bien défini. Ce typage fort aide à se prémunir de bugs ou de petites étourderies en vérifiant automatiquement la compatibilité des types. En revanche, contrairement au C, le typage n'est pas explicitement écrit à chaque déclaration, OCaml arrive à l'inférer automatiquement.

| **Symbole** | `+.` | `-.` | `*.` | `/.` | `**` |
|---|---|---|---|---|---|
| **Signification** | Addition flottante | Soustraction flottante | Multiplication flottante | Division flottante | Puissance flottante |

À noter que l'opérateur de puissance flottante `**` existe, mais qu'il ne s'agit pas de la puissance entière et ne marche donc pas avec des entiers. Il est également possible d'utiliser la notation scientifique : par exemple `3e7` pour le flottant de valeur $3 \times 10^7$. Il faut bien retenir que la valeur n'est pas de type `int`, même si cela est a priori un entier.

**10** Entourer les déclarations globales correctes qui ne produiront pas d'erreurs lors de leur évaluation. Corriger celles qui présentent une erreur manifeste. Ce n'est pas toujours une erreur de typage.

```
let pi = 3.14 + 0.001
```

```
let p = 2 + 4
```

```
let g = 4.0 ** 5
```

```
let h = 4. +. 3.0
```

```
let s1 = 2.0 **. 16.
```

```
let PI = 3.14 +. 0.
```

```
let i = p + h
```

```
let f1 = 10.0 /. 2
```

Lorsque cela est absolument nécessaire, il reste possible de convertir la valeur d'un entier en flottant et vice versa. On utilise pour cela `float_of_int` ou `int_of_float` respectivement, suivi d'un espace puis de la valeur à convertir.

**11** Prédire ce que va afficher l'interpréteur pour ces déclarations globales, puis vérifier en pratique :

```
let j = float_of_int 7 +. 3.
```

```
let k = 8 + int_of_float 1e2
```

```
let n = int_of_float float_of_int 7
```



## IV. Parenthésage d'expressions

Le dernier exemple provoque une erreur, car la fonction `int_of_float` attend un flottant, alors que ce qui est écrit après est une fonction (`float_of_int`). Cela n'est pas complètement aberrant : en OCaml, il est a priori tout à fait possible qu'une fonction en prenne une autre en argument. Cela rejoint l'esprit du **paradigme fonctionnel**, dans lequel les fonctions sont des objets de première classe, que l'on va pouvoir manipuler plus librement qu'en C.

Nous aurons l'occasion de découvrir cela plus tard : ici il est clair que l'on cherchait à évaluer `float_of_int 7` en premier avant d'appliquer la fonction `int_of_float` sur la valeur obtenue. Pour cela, il est possible d'utiliser des parenthèses pour prioriser certaines évaluations :

```
let n = int_of_float (float_of_int 7)
```

```
let x = (3 * (4 + 1) / (1 + 1)) mod 2
```

### Règle à retenir

En OCaml, on évitera tant que possible de mettre des parenthèses inutiles. Il est notamment courant qu'un appel de fonction s'écrive `f x` et non `f (x)` ou `f(x)` comme en C. En revanche, on s'autorise à les mettre s'il est **nécessaire** de prioriser une évaluation, ou si la lisibilité du code s'en trouve améliorée.

## V. Le type bool

Le langage OCaml offre directement la possibilité de travailler avec des booléens. Tout comme en C (et contrairement à Python), les deux booléens sont notés `true` et `false` **avec une minuscule**. Il est possible de les manipuler avec les opérateurs booléens suivants, qui sont tous à connaître et à ne surtout pas confondre avec ceux du langage C :

| **Symbole** | `=` | `<>` | `>` ou `<` | `>=` ou `<=` | `not` | `&&` | `\|\|` |
|---|---|---|---|---|---|---|---|
| **Signification** | Égalité | Inégalité | Comparaison stricte | Comparaison large | Négation (NON) | Conjonction (ET) | Disjonction (OU) |

### Remarques et pièges classiques à éviter

- l'égalité de deux valeurs se note avec un simple égal (`=`), et non un double égal (`==`) comme en C ou en Python.
- l'opérateur d'égalité est le même pour les entiers, pour les flottants, et pour tous les types simples. **Même pour les chaînes de caractères** ce qui est faux en C (où les chaînes de caractères ne sont pas un type simple...) !
- les opérateurs `==` et `!=` existent bien en OCaml, mais ont une autre signification : l'**égalité (ou inégalite) physique**, c'est-à-dire tester s'il s'agit du même endroit dans la mémoire ou pas.

**12** Prédire ce que va afficher l'interpréteur pour ces déclarations globales, puis vérifier en pratique. Attention à ne pas confondre le symbole `=` qui sert à la déclaration globale, de celui qui est l'opérateur booléen d'égalité entre deux valeurs.

```
let a = 12
```

```
a = 108
```

```
let g = (a = a) = (a = a)
```

```
let t = 0.1 +. 0.2 = 0.3
```

```
let h = a = a = a = a
```

```
let i = not (a >= 2) || true
```

On peut fermer l'interpréteur en écrivant `exit 0;;` puis **Entrée**.