


# I. OCaml : un langage compilé

## Compiler un fichier source OCaml

```
ocamlc tp13.ml -o tp13
```

```
ocamlopt tp13.ml -o tp13
```

puis exécuter le fichier ainsi produit :

```
./tp13
```

OCaml est un langage interprété, mais c'est aussi un langage **compilé**. Il existe deux compilateurs principaux à connaître :

Le premier, **ocamlc**, compile le code en **bytecode**, un code intermédiaire qui sera ensuite exécuté par un programme. C'est un bon choix par défaut quand on souhaite une compilation rapide.

Le second, **ocamlopt**, compile le code en langage machine directement. Cela produit un programme plus rapide à l'exécution, mais peut demander plus de temps à la compilation.

La syntaxe permettant de compiler un code source OCaml est très similaire à celle permettant de compiler un code C.

**1** Créer un nouveau fichier entièrement vide et le nommer **tp13.ml**. Tester que ce fichier compile bien avec les commandes ci-dessus. Par la suite, il sera demandé de régulièrement compiler votre code.

## Commentaires en OCaml

Un commentaire en OCaml commence par `(*` et se termine par `*)`. Voici trois exemples :

```
(* TP 13 *)
```

```
(* Un commentaire *)
```

```
(* TODO *)
```

**2** Écrire un commentaire dans votre fichier **tp13.ml**, et vérifier que tout compile bien.
A-t-on le droit d'écrire un commentaire sur plusieurs lignes ?

# II. Le type unit

**3** Dans **tp13.ml**, écrire la déclaration globale `let a = 8 + 2 * 50`. Il n'y a pas besoin des deux points-virgules après la liaison ici, car on ne va pas chercher à interpréter le code mais uniquement à le compiler. Vérifier que le code ainsi écrit compile bien. Que se passe-t-il à l'exécution ?

Un programme OCaml va parcourir le fichier de haut en bas, et pour chaque déclaration globale il va évaluer l'expression et la lier à l'identifiant. La valeur de l'expression `8 + 2 * 50` a donc bien été calculée et liée à l'identifiant `a`, mais rien n'a demandé d'afficher cette valeur. Le programme se termine donc sans erreur mais aussi sans affichage.

Il est possible d'utiliser des fonctions permettant d'afficher quatre des cinq types simples vus dans le TP précédent :

| **Fonction** | `print_string` | `print_int` | `print_float` | `print_char` |
|---|---|---|---|---|
| **Affiche...** | une chaîne de caractères | un entier | un flottant | un caractère |

Mais quel est le type de `print_int a` ? On sait que `a` est de type `int`, mais la fonction d'affichage se contente d'afficher la valeur par effet de bord. Pourtant, en OCaml tout doit avoir un type. On donne alors le type `unit`. Comme la valeur à lier ne nous intéresse pas, on utilise la notation `_` ou `()` à la place d'un identifiant pour indiquer explicitement que nous étions uniquement intéressés par l'effet de bord, pas par la valeur récupérée qui sera `()` de type unit.

**4** Tester la fonction d'affichage en ajoutant la ligne `let () = print_int a` dans votre fichier. Que se passe-t-il lors de l'exécution ?

**5** Ajouter deux lignes pour afficher deux autres valeurs de votre choix de types différents, par exemple la chaîne de caractères `"Hello CIV !\n"`. Que se passe-t-il si l'on tente d'afficher une valeur du mauvais type ?

## Deux vieilles connaissances

- Pour des affichages plus complexes, il est également possible d'utiliser la fonction d'affichage `Printf.printf` qui fonctionne de la même manière qu'en C, avec une chaîne de format et des valeurs à insérer. Dans un appel de fonction à plusieurs arguments, il n'y a pas besoin de les séparer par une virgule. On peut par exemple écrire :

```
let () = Printf.printf "La valeur de a est %d et son carré vaut %d\n" a (a * a)
```

- Lorsque l'on souhaite réaliser un test, on peut utiliser librement la fonction `assert`, qui attend un booléen et lève une exception si jamais il se trouve être faux. On peut par exemple écrire la déclaration globale :

```
let () = assert (a = 108)
```

**6** Ajouter ces deux déclarations globales dans votre fichier et compiler. Les parenthèses sont-elles nécessaires ici ? Tester en les enlevant.

On retiendra que l'appel de fonction est prioritaire sur les autres opérations en OCaml. Par exemple la déclaration globale `let () = print_string 1 + 1` n'est pas correcte, car l'évaluation commence d'abord par l'appel de fonction, c'est-à-dire `let () = (print_string 1) + 1` ce qui provoque une erreur de typage.

# III. Expressions conditionnelles

## Expression conditionnelle

Une **expression conditionnelle** est une **expression** de la forme `if cond then expr_1 else expr_2`, avec :
- `cond` qui doit être une expression booléenne
- `expr_1` et `expr_2` qui doivent être deux expressions **de même type**.

Il est à noter qu'on parle ici bien d'une **expression** conditionnelle, ce n'est pas tant une structure de branchement comme en C, mais bien une expression ayant un type et pouvant être utilisée dans d'autres expressions. C'est pour cela que `expr_1` et `expr_2` doivent être de même type, qui sera aussi le type de l'expression conditionnelle. Par ailleurs, à chaque `then` doit obligatoirement être associé un `else`. Sinon le else par défaut renvoie `()` de type unit, et cela provoque une erreur de typage sauf si l'expression du then est aussi de type unit.

**7** Que vaut l'expression `if a = 108 then 4.0 else 10.0` ? Vérifier en la liant à un identifiant puis en affichant la valeur de ce dernier.

**8** Parmi les expressions suivantes, entourer celles qui sont correctes et prédire leur valeur, puis réfléchir à une éventuelle manière de corriger les autres. Vérifier les valeurs en compilant.

```
if (1 + 1 == 2) then 4 else 3.0
```

```
3 + (if a = 108 then 2 + 3 else 0) + 41
```

```
if a = 108 then 108 + 108 + 108
```

```
if (if true then false else true) then true else false
```

# IV. Liaisons locales en OCaml

Lorsqu'une déclaration globale est faite, la portée de la variable ainsi définie correspond à tout le reste du code (toutes les lignes en dessous peuvent accéder à cet identifiant pour former de nouvelles expressions). Il est possible de limiter cette portée en définissant des **liaisons locales**, pour des variables locales qui n'existent que pour évaluer l'expression actuelle mais pas par la suite. On utilise pour cela la syntaxe `let <id> = <expr> in`, c'est-à-dire exactement comme pour une déclaration globale mais avec le mot-clef `in` à la fin, qui indique que ce n'est pas terminé et que cette valeur liée servira à l'évaluation d'une expression qui arrive ensuite uniquement. Tout comme pour les liaisons globales, une liaison locale peut être masquée si une nouvelle liaison se produit avec le même identifiant.

**9** Que valent les expressions suivantes ? Vérifier en les liant dans des déclarations globales et en affichant la valeur obtenue.

```
let a = 108 in
let b = a + a in
a + b
```

```
let a = 1 in
let b = 2 + if a = 1 then 0 else 4 in
if (a + b) > 4 then b else a
```

```
let a = 2 in
let a = 3 in
a + a
```

```
let a = if (let a = 1 in a = 0) then (let b = 1 in 2) else 4 in
let b = if a / 2 > 1 then 1.0 else 2.0 in
b
```

# V. Définir des fonctions en OCaml

En OCaml, les fonctions sont des objets de première classe que l'on va pouvoir utiliser de nombreuses manières. Définir une fonction revient en fait à réaliser une déclaration globale entre un identifiant (le nom de la fonction), et une expression de fonction. Pour indiquer clairement cela, il y a deux syntaxes à connaître :

```
let <id> = fun <arg> -> <expr>
```

Exemple :

```
let carre = fun x -> x * x
```

```
let entier_suivant = fun n -> (n + 1)
```

```
let <id> <arg> = <expr>
```

Exemple :

```
let carre x = x * x
```

```
let entier_suivant n = n + 1
```

Dans le premier cas, on reste dans le style classique des déclarations globales faites jusqu'à maintenant. Ce qui est à droite du signe égal est une **expression fonctionnelle**, c'est-à-dire une expression dont la valeur est une fonction (ici la fonction carré). Dans le second cas, on utilise une syntaxe alternative plus proche de l'écriture mathématique.

Dans le cas où l'on souhaite donner plusieurs arguments, comme par exemple pour la fonction qui réalise la somme de deux entiers, la seconde forme se généralise aisément en les ajoutant après l'identifiant de fonction, séparés par des espaces. Pour la première forme, on peut construire une fonction à plusieurs arguments comme un assemblage de fonctions à un seul argument, chacune prenant un argument renvoyant la fonction suivante.

Exemple :

```
let somme = fun x -> fun y -> x + y
```

Exemple :

```
let somme x y = x + y
```

## Signature d'une fonction

La **signature** d'une fonction contient son nom et décrit son type : elle précise les types des valeurs qu'elle prend en argument (à gauche) ainsi que le type de la valeur qu'elle renvoie (à droite). Par exemple la fonction `carre` est de signature `carre : int -> int`, qui se lit "la fonction `carre` prend un entier en argument et renvoie un entier". La signature de la fonction `somme` est `somme : int -> int -> int`, les deux premiers `int` correspondent aux types des deux arguments `x` et `y`, et le dernier (à droite) au type de la valeur renvoyée.

**10** Implémenter une fonction de signature `cube : int -> int` qui calcule le cube d'un entier, avec les deux syntaxes proposées. La tester sur un exemple.
Que se passe-t-il si l'on écrit `let a = cube -1` ? Pourquoi et comment corriger cela ?

**11** Implémenter une fonction de signature `max_trois : int -> int -> int -> int` qui à partir de trois entiers s'évalue pour donner le plus grand des trois. Utiliser les deux syntaxes proposées.

**12** Implémenter une fonction de signature `implique : bool -> bool -> bool` qui implémente l'implication mathématique ($\Rightarrow$) sous forme de fonction booléenne.

**13** Réfléchir à ce que renvoie `somme 10`. Compiler et tester. Nous aurons l'occasion de discuter de cela plus en détail au prochain TP.