TP n°7 – Découverte de OCaml

1 Compilation et interprétation

Ocaml est un langage qui peut être interprété ou compilé. Un fichier de code Ocaml a comme extension `.ml`.

- Pour compiler : taper `ocamlc -o fichier.out fichier.ml` pour compiler (c'est comme C sauf qu'on utilise un autre compilateur) puis `./fichier.out` pour exécuter.
  Comme en C rien ne s'affichera sauf si vous avez utilisé une fonction d'affichage (comme `Printf.printf`).
- Pour interpréter vous avez deux choix :
  - `ocaml fichier.ml` quand vous débugguez. Cela affiche la première erreur ou rien s'il n'y a pas d'erreurs.
  - `ocaml < fichier.ml` pour interpréter tout le fichier à la fois.
    Cela affiche dans le terminal le type et le résultat de **chaque ligne** et les affichages fait avec `Printf.printf`.
    Attention : débugguer de cette manière est pénible, les messages d'erreurs étant parfois partiellement effacés.

Remarque : Vous pouvez coder sous VSCode comme en C. Par défaut il n'y aura pas de couleur. Pour obtenir les couleurs, il faut d'abord vous connecter à internet : ouvrir firefox, se connecter avec les identifiants des ordis du reste du lycée.

Ensuite, aller dans l'onglet extensions de VScode et installer ocaml-platform

2 Les bases

Dans cette partie on propose des expressions en Ocaml mais pas leur résultat. **Pour chaque expression, vous devez tester avec Ocaml pour comprendre ce qu'elle fait**.

Les parties entre `(*` et `*)` sont des commentaires Ocaml, pas du code.

1. Les expressions

Le code Ocaml est divisé en **expressions** qui se terminent toujours par `;;`. Chaque expression a un type et une valeur, auquels on peut accéder en interprétant l'expression.

Par exemple :

OCaml

```
2+2;; (* Calcul, type int et valeur 2 *)
let z = 35;; (* Liaison, type int et valeur 35 *)
let f x = 35;; (* Fonction, type int -> int et valeur <fun> *)
```

Les types de base en OCaml sont les types entier `int`, flottant `float`, booléen `bool`, caractère `char` et chaîne de caractères `string`. **Contrairement à C, on a pas besoin de donner le type, c'est Ocaml qui le devine**.

On peut former des types composés comme `int -> int` indiquant une fonction des entiers vers les entiers ou `int * int` indiquant un couple d'entiers.

2. Le type `int`

Les entiers sont représentés par le type `int`, on dispose des opérateurs `+`, `-`, `*`, `/` et `mod`.

1. Tester les opérations suivantes et observer le résultat. Quelle opération arithmétique réalise l'opérateur `/` ? Quelle opération arithmétique réalise l'opérateur `mod` ?

```OCaml
1 + 1;;
3 - 5;;
2 * 10;;
10 / 3;;
10 mod 3;;
```

2. Créer une variable `x` valant 3 et une variable `y` valant 6. Effectuer l'addition des deux.

3. Le type `float`

Comme en C on a le type `float` pour représenter les flottants :

```OCaml
let f1 = 1.64;;
let f2 = 1.;;
```

Les opérations précédentes sur les entiers ne peuvent pas s'appliquer sur des flottants. On dit que OCaml est un langage fortement typé. Il faut utiliser les opérateurs sur les flottants qui sont obtenus en ajoutant un point à la fin comme suffixe : `+.`, `-.`, `*.`, `/.`. On dispose de plus de l'opérateur d'exponentiation `**` (il n'y a pas de point mais c'est une opération de flottants).

3. Additionner les variables `f1` et `f2`. Si vous avez une erreur, vérifiez que vous avez lu le paragraphe précédent.
4. Si on écrit `let f3 = 6;;`, quel est le type de `f3` ? Comment faire pour que ce soit un flottant ?
5. Calculer 6^3.
6. Détecter et corriger les erreurs dans les calculs suivants :

```OCaml
2.0 + 2.0;;
1. / 1.;;
2. +. 2;;
2.0 ** 3;;
(2.0 + 1.) *. (3. ** 3.);;
2 ** 10;;
```

Pour Ocaml, 10 et 10.0 sont fondamentalement différents. On peut l'aider à traduire l'un vers l'autre avec une instruction **explicite** qu'on appelle une coercition (ou cast). À l'inverse, C fait des coercitions implicites (à la limite en mettant un warning).

```OCaml
int_of_float 10.;;
float_of_int 10;;
```

7. Testez `int_of_float 3.4` et `int_of_float 3.6`. Quel est l'arrondi pratiqué par OCaml ?
8. En déduire si on a toujours `float_of_int (int_of_float x)` égal à `x` ?

Les fonctions mathématiques usuelles sont définies pour les flottants, mais pas les entiers. Par exemple `sin`, `cos`, `tan`, `acos`, `asin`, `atan`, `exp`, `log` (le logarithme népérien), `log2`, `log10`, `sqrt` (la racine carrée), ...

```OCaml
let f4 = cos 0.5;;
let f5 = exp (f1+.f2);;
```

Remarque 2.1. **Très important!!!!** En Ocaml il n'y a pas de parenthèses obligatoires autour des arguments de fonctions. En revanche si l'argument est résultat d'une opération, il faut mettre des parenthèses, comme dans `f5` juste au-dessus.

4. Le type `bool`

Le type booléen `bool` de Ocaml est très similaire à celui de C, il prend deux valeurs `true` et `false`. Le "et" et le "ou" s'écrivent `&&` et `||`. En revanche de le "non" logique s'écrit `not`, par exemple `not (true && false)`.

Pour les comparaisons, on peut utiliser `<`, `>`, `<=`, `>=`, `=` (test d'égalité), `<>` (test de différence). **Attention aux différences avec le C**.

OCaml

```
true;;
((not true) || false) && (false && true);;
3 > 3;;
(2 = 2) || (3 <> 4) || (1 <= 2) || (2 >= 1);;
2.2 > 2.1 && 2.1 > 2.0;;
3.0 *. 0.1 = 0.3;;
```

9. Tester le test `2>2.3;;`. Peut on comparer des entiers et des flottants ?

Remarque : les opérateurs de comparaisons sont valables pour tous tous les types tant que les deux expressions comparées sont de même type. On dit qu'ils sont **polymorphes**.

5. Le type `char`

OCaml dispose du type `char` pour les caractères notés entre **guillemets simples** et on dispose de conversions depuis et vers les codes ASCII (code entier correspondant à un caractère selon la table ASCII) :

OCaml

```
int_of_char '\n';;
char_of_int 32;;
```

10. Quel est le numéro dans la table ASCII de ';'? De 'O'?

6. Le type `string`

Contrairement à C il existe un type prédéfini pour les chaînes de caractères, le type `string`, entre double guillemets :

OCaml

```
"Hello world!";;
```

L'opérateur ^ permet de concaténer deux chaînes.

OCaml

```
"Hello" ^ " " ^ "world" ^ "!"= "Hello world!";;
```

7. Liaisons globales

Pour créer une variable globale en Ocaml, on utilise une liaison globale, c'est à dire une liaison en dehors de toute fonction ou instruction.

OCaml

```
let y = "y";;
```

Remarque très importante : une variable en Ocaml ne se modifie pas. Elle est créée avec une valeur qui est immuable.

8. Liaisons locales

On peut créer une variable locales, dont la portée est ce qui est compris entre le mot-clé `in` et le prochain `;;`

```OCaml
let x = 0 in x * x;;
```

11. Tester le code suivant et expliquer ce qui se passe :

```OCaml
let y = 12 in 3*y;;
y;;
```

12. Combien y a t'il de liaisons globales dans le code suivant ? De liaisons locales ?

```OCaml
let trente =
  let quinze =
    let cinq = 2 + 2 + 1 in
    let deux = 1 + 1 in
    let sept = cinq + deux in
    cinq + deux + sept + 1
  in
  quinze + quinze
;;
```

9. Types produits

Comme en Python (et pas comme en C), on peut construire des paires et des tuples. Observez bien les types obtenus pour chacune des lignes suivantes, des `*` apparaissent :

```OCaml
(1, 2);;
("Zéro", '0', 0, 0.);;
(2, 3) = (1 + 1, 3);;
(2.0, 1) = (2, 1.0);; (*Ici il y a une erreur vdans la comprendre et corriger. Regardez bien les types*)
```

10. `let` destructurant

La liaison `let` peut également être utilisée pour « déconstruire » un tuple, c'est le moyen pour accéder aux champs d'un tuple :

```OCaml
let paire = (3, 2);;
let (x, y) = paire;;
```

Lorsque l'un des champs ne nous intéresse pas, on peut utiliser la variable spéciale _ qui sert de poubelle.

```OCaml
let coordonnees = (3.0, 2.0, 10.0);;
let (abscisse, _, _) = coordonnees;;
```

13. Écrire un code qui récupère le deuxième et le quatrième élément de `(2,4,5,12)` et les multiplie entre eux.

11. Fonctions

On peut définir une fonction à l'aide du mot clef `function` :

```OCaml
let f = function x -> x - 1;;
```

On peut aussi utiliser la syntaxe équivalente :

```OCaml
let f x = x - 1;;
```

Ces deux bouts de code définissent la fonction f qui à x associe $x-1$ et qui va des entiers dans les entiers.

14. Utiliser les deux manières pour définir $f : x \mapsto x^2 + 3 * x + 1/2$ qui a un flottant associe un flottant. (N'oubliez pas de mettre des `.` partout)

Dans l'exemple précédent on avait qu'un seul argument. Quand on a plusieurs arguments, le mot-clé **`function`** ne marche plus. On utilise donc la deuxième forme, mais on a plusieurs syntaxes possibles

```OCaml
let produit (x,y,z) = x*y*z;; (* Utiliser un tuple *)
let produit x y z = x*y*z;; (* Ne pas utiliser un tuple *)
```

15. Observer la différence entre les types des deux fonctions ci-dessus. Arrivez-vous à repérer le type du tuple ?
    On y reviendra, mais en général quand on vous demandera d'écrire une fonction Ocaml, le type sera précisé et c'est à vous d'identifier si un tuple est nécessaire.
16. Écrire une fonction **`somme : int -> int -> int -> int`** qui fait la somme de trois entiers.

12. Conditionnelles

Une expression conditionnelle s'écrit `if expr1 then expr2 else expr3`. Bien entendu **`expr1`** doit être de type **`bool`** et **`expr2`** et **`expr3`** doivent être de même type. Le type de cette expression conditionnelle est le type mutuel des expressions **`expr2`** et **`expr3`**.

Sa valeur est celle de **`expr2`** si la valeur de **`expr1`** est **`true`** et celle de **`expr3`** si la valeur de **`expr1`** est **`false`**.

```OCaml
let grrr = if "oui" = "non" then "o" else 'n';; (*Ici erreur car le then et le else ne sont pas du même type*)
let abs = function x -> if (x > 0.) then x else -.x;;
let abs x = if (x > 0.) then x else -.x;;
```

17. On considère l'expression conditionnelle suivante :

    ```OCaml
    if let x = 42. in x *. x > 1700. then
      if "Caml" > "Python" then 12
      else 21
    else if true then 0
         else let y = 1 in 1 - y * y
    ;;
    ```

    Identifiez **`expr1`**, **`expr2`** et **`expr3`**. N'hésitez pas à ajouter des parenthèses.

3 Exercices

Exercice 1 Petites fonctions

1. Écrire une fonction **`carre : int -> int`** qui calcule le carré d'un entier donné en entrée.
2. Écrire une fonction **`implique : bool * bool -> bool`** qui étant donné un couple de booléens (a, b) calcule $a \Rightarrow b$.
3. Écrire une fonction **`division_euclidienne : int -> int -> int * int`** qui étant donné deux entiers a et b renvoie le couple formé du quotient et du reste de la division euclidienne de a par b.

Exercice 2 Ou logique

L'expression `e1 && e2` signifie `if e1 then e2 else false`. Traduisez de même `e1 || e2` à l'aide d'une expression conditionnelle.

Exercice 3 Manipuler les flottants

1. En utilisant uniquement des liaisons **locales**, calculer $\dfrac{1+\sqrt{2}+\sqrt{2}^3}{e^{\sqrt{2}}-1}$.
2. Écrire une fonction `th : float -> float` qui calcule la fonction tangente hyperbolique tanh : $x \mapsto \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$.

Exercice 4 Manipuler les caractères

1. Définir deux fonctions `est_minuscule` et `est_majuscule` qui testent respectivement si le caractère ASCII passé en entrée est une lettre minuscule et une lettre majuscule. Quel doit être leur type de ces fonctions ?
   On pourra consulter sur internet la table ASCII et utiliser la fonction `int_of_char`.
2. Définir la fonction `majuscule : char -> char` qui renvoie la version majuscule d'une lettre si son argument est une lettre minuscule et renvoie le caractère inchangé sinon. On pourra utiliser `char_of_int`.

Exercice 5 Min et Max

Écrire les fonctions `mini : 'a -> 'a -> 'a` et `maxi : 'a -> 'a -> 'a` polymorphes qui renvoient respectivement le plus petit et le plus grand de leurs arguments.

Remarque Le `'a` dans les types est une variable de type, qui peut être remplacée par n'importe quel type. Cela signifie que les fonctions `mini` et `maxi` fonctionnent peu importe le type d'entrée, tant que les deux entrées sont du même type.

Un autre exemple serait une fonction `f : 'a -> 'b -> 'b`. Ses entrées peuvent être de n'importe quel type et pas forcément le même. En revanche le type de la sortie est le même que celui de la 2ème entrée.

Vous n'avez rien à faire de particulier pour que les fonctions `mini` et `maxi` aient ce type. Vous pouvez les écrire comme si vous manipuliez des entiers ou des `string` et ça marchera.

Remarque Les fonctions `min` et `max` sont prédéfinies en OCaml et vous pourrez dorénavant les utiliser.

Exercice 6 Manipuler les couples

Voici plusieurs manières équivalentes de définir la fonction `fst : 'a * 'b -> 'a` (`fst` pour first) qui renvoie le premier élément d'un couple :

OCaml
```
let fst = function (a, b) -> a;;
let fst = function (a, _) -> a;;
let fst (a, b) = a;;
let fst (a, _) = a;;
let fst couple =
  let (a, b) = couple in
  a
;;
let fst couple =
  let a, b = couple in
  a
;;
let fst couple =
let a, _ = couple in
  a
;;
```

Définir de toutes ces façons équivalentes, la fonction `snd` (pour second) qui renvoie le deuxième élément du couple. Commencez par prévoir son type.

Remarque Les fonctions `fst` et `snd` sont prédéfinies en OCaml et vous pouvez les utiliser. Il est cependant aussi simple d'écrire `let x, _ = couple` que `let x = fst couple`.

Exercice 7 Manipuler les tuples

Implémenter, après avoir deviné leur type, les fonctions suivantes :

1. La fonction `duplicate` qui sur un argument `a` s'évalue en le couple `(a, a)`.
2. La fonction `swap` qui sur un couple `(a, b)` s'évalue en le couple `(b, a)`.
3. Une fonction `concat` qui sur deux couples `(a, b)` et `(c, d)` s'évalue en le quadruplet `(a, b, c, d)`.