Informatique *Cours*
Programmation fonctionnelle avec OCaml Ipesup

Objectifs

- Utiliser OCaml pour manipuler des entiers et des flottants.
- Comprendre les déclarations, variables et fonctions.
- Écrire un premier programme simple.

1 OCaml

OCaml, conçu à l'Inria dans les années 1980, est un langage *multi-paradigmes* qui combine programmation impérative (boucles, variables modifiables, comme en C ou Python) et programmation orientée objet (comme en Java).

Il se distingue surtout par la *programmation fonctionnelle*, où les variables sont immuables : elles sont définies une fois pour toutes et ne peuvent plus être modifiées. Dans ce cadre, les boucles classiques cèdent la place aux fonctions *récursives*, et les fonctions *d'ordre supérieur* permettent de passer des fonctions en argument ou de les retourner, ce qui favorise modularité et réutilisation.

OCaml intègre néanmoins des mécanismes impératifs (boucles, variables modifiables, exceptions), ce qui facilite le mélange de différents styles dans un même langage.

On commence par manipuler des programmes simples où les données de base sont des entiers et des booléens.

2 Déclarations globales et expressions simples

Un programme OCaml est composé d'une suite de *déclarations*. Chaque déclaration associe un identificateur à une expression. La syntaxe est la suivante :

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

`<id>` est un identificateur, c'est-à-dire une séquence de lettres (sans accents), de chiffres ou du symbole. Un identificateur doit toujours commencer par une lettre minuscule.
`<expr>` est une expression arithmétique, booléenne ou plus complexe.

Exemple

```
let x = 42
```

Cette déclaration associe la valeur `42` au nom `x` et enregistre cette information dans l'*environnement global*.
On peut ensuite utiliser cette variable dans d'autres expressions :

```
let y = x * x + 10
```

La valeur associée à `y` est alors $42 \times 42 + 10 = 1774$.

Une déclaration définit toujours une nouvelle variable, ainsi :

```
let x = 100
```

ne modifie pas la première variable `x = 42`. La nouvelle définition masque simplement l'ancienne. C'est ce qu'on appelle la *portée lexicale*. Une variable reste liée à la valeur définie dans son environnement, indépendamment des redéfinitions suivantes.

Exemple

```
let x = 42
let y = x * x + 10
let x = 100
```

Dans l'expression `x + 100`, la variable `x` désigne encore la première définition (`x = 42`). La portée de la première définition ne se termine qu'au moment où la deuxième déclaration est entièrement évaluée.

Il est souvent nécessaire d'afficher une valeur. On peut utiliser :

```
let () = Printf.printf "%d\n" y
```

Cette écriture permet d'imprimer le contenu de `y`.

3 Interprétation et compilation d'un programme

Un programme OCaml doit être enregistré dans un fichier avec l'extension `.ml`. Par exemple, le fichier `premier.ml` peut contenir :

```
let x = 42
let y = x * x + 10
let () = Printf.printf "%d\n" y
```

Il existe plusieurs façons d'exécuter un programme OCaml, soit directement depuis le **terminal**, soit en utilisant des **environnements de développement** (éditeurs/IDE).

Depuis le terminal

— **Exécution directe avec l'interpréteur**

```
$ ocaml premier.ml
1774
```

Lit et exécute le fichier ligne par ligne, puis termine. Utile pour tester rapidement un petit programme.

— **REPL interactif**

```
ocaml
# let x = 42 ;;
val x : int = 42
```

Ouvre une session interactive (Read-Eval-Print Loop). Idéal pour expérimenter des expressions et vérifier les types. Variante plus confortable : `utop`.

— **Compilation Bytecode avec `ocamlc` :**

```
$ ocamlc -o premier premier.ml
$ ./premier
1774
```

Produit un exécutable portable qui fonctionne sur toutes les machines ayant l'environnement OCaml installé. Démarrage rapide, mais vitesse d'exécution un peu plus lente.

— **Compilation Natif avec `ocamlopt` :**

```
$ ocamlopt -o premier premier.ml
$ ./premier
1774
```

Produit un exécutable compilé en code machine, donc plus rapide à l'exécution. C'est l'option à privilégier quand on veut distribuer ou utiliser un programme de manière efficace.

Avec un éditeur ou une application

— **VS Code + extension OCaml Platform** Fournit la coloration syntaxique, l'affichage des erreurs en direct, la navigation dans le code et l'intégration avec `dune`.

— **Emacs (mode Tuareg) + Merlin** Environnement historique pour OCaml, puissant pour l'édition, l'autocomplétion et la gestion des types.

— **Try OCaml (en ligne)** : `https://try.ocamlpro.com/` Permet de tester du code OCaml directement dans le navigateur, sans installation.

— **Dune (outil de build recommandé)** Gestionnaire officiel de projets OCaml. Exemple :

```
dune init exe premier .
dune exec ./premier.exe
```

Simplifie la compilation et l'organisation de projets plus grands.

4 Inférence de types

En OCaml, il n'est pas nécessaire d'indiquer explicitement le type d'une variable lors de sa déclaration. Cela ne veut pas dire que les variables n'ont pas de type : au contraire, OCaml est un langage *fortement typé*, où chaque expression et chaque variable possède un type unique, déterminé dès sa définition.

La détermination du type est faite automatiquement par un algorithme d'*inférence de types*. Cet algorithme analyse les expressions et calcule leur type sans demander d'annotation de la part du programmeur.

Exemple

```
let x = 42
let y = x * x + 10
```

Ici, OCaml déduit tout seul que `x` et `y` sont de type `int`.

Même si cela paraît simple dans cet exemple, l'inférence peut gérer des expressions bien plus complexes. Cette automatisation rend l'écriture plus concise tout en maintenant une grande rigueur dans la gestion des types.

Un avantage majeur de cet algorithme est qu'il détecte les incohérences de typage à la compilation. Si une expression contient une erreur de type, le compilateur refuse de produire un exécutable. Cela permet d'éviter de nombreux *bugs* à l'exécution.

Ainsi, un programme bien typé en OCaml a de fortes garanties de sûreté, bien plus qu'un programme écrit dans un langage permissif (comme C) ou faiblement typé (comme Python).

5 Expressions simples

Les expressions en OCaml couvrent plusieurs catégories de valeurs : entiers, flottants, booléens, caractères et chaînes de caractères.

Nombres entiers relatifs

Les entiers s'écrivent comme des suites de chiffres, avec la possibilité d'utiliser le caractère souligné pour améliorer la lisibilité (`1_000_000` pour un million).
Les opérateurs arithmétiques disponibles sont :

- addition : `+`,
- soustraction : `-` (unaire ou binaire),
- multiplication : `*`,
- division euclidienne : `/`,
  attention : une division par zéro provoque une exception `Division_by_zero`.
- reste de la division : `mod`.

```
let x = 5_234_456 + 2 * 67
let y = -5 + x / 4
let z = x mod 5 + 190
```

Sur un processeur à n bits, OCaml utilise $n-1$ bits pour représenter un entier relatif, le dernier bit étant réservé au signe. L'intervalle de valeurs disponibles est donc restreint, et les valeurs extrêmes sont accessibles via `min_int` et `max_int`.
L'arithmétique entière est dite *modulaire*, car l'opération suivante est toujours vraie :

```
max_int + 1 = min_int
```

Nombres décimaux.

Les nombres à virgule (ou flottants) peuvent s'écrire de deux façons :

- la notation classique avec un point comme séparateur décimal,
- la notation scientifique, où un flottant est décomposé en trois parties :
  - le signe $(+)$ ou $(-)$,
  - la mantisse m, qui représente la partie significative,
  - l'exposant n, qui détermine le décalage de la virgule.

  Ainsi, `1.23e3` signifie $1.23 \times 10^3 = 1230$ et `4.5e-2` vaut $4.5 \times 10^{-2} = 0.045$.

```
let x = 3.141_592
let y = 0.232e-4
let z = -45.897E7
```

Les flottants sont codés sur 64 bits selon la norme IEEE 754 double précision. Leur type est `float`.
Les opérateurs arithmétiques sont distincts de ceux des entiers :

- `+.` pour l'addition,
- `-.` pour la soustraction,
- `*.` pour la multiplication,
- `/.` pour la division,
- `**` pour l'exponentiation.

```
let u = (x /. z +. 5.4) *. y
```

Booléens.

OCaml propose deux valeurs booléennes : `true` et `false`. Les opérateurs disponibles sont :

- négation : `not`,
- conjonction : `&&`,
- disjonction : `||`.

```
let b = (x > 0 && not y) || (not x && y)
```

Ces opérateurs sont *paresseux* : une partie de l'expression n'est évaluée que si nécessaire.
Le type associé est `bool`.

Caractères et chaînes de caractères.

Le type des caractères est `char`, et celui des chaînes est `string`. Chaque caractère est associé à un code numérique selon la table ASCII (0–255).
Certains caractères spéciaux s'écrivent avec des séquences d'échappement :

- `'\n'` : saut de ligne
- `'\r'` : retour chariot
- `'\t'` : tabulation
- `'\\'` : antislash
- `'\"'` : guillemet droit

Les chaînes de caractères sont délimitées par des guillemets doubles.

```
let s = "Bonjour le monde"
```

L'opérateur `^` permet de concaténer deux chaînes.
Il est aussi possible d'accéder à un caractère précis grâce à l'indexation : `s.[i]`, où le premier caractère est à l'indice 0.

Exemple

```
let s1 = "Hello !\n"
let s2 = "The symbol \" is in a string"
let s = s1 ^ s2 ^ "!!\n"
let c = s.[2]
```

Ici, `c` contient le caractère `l`.

Un caractère `'a'` est différent d'une chaîne `"a"`. Le premier est un `char`, stocké sur un octet ; la seconde est une `string`, c'est-à-dire une séquence de caractères. Les chaînes sont immuable : on ne peut pas modifier un caractère à l'intérieur. Les comparaisons sur les chaînes utilisent l'ordre lexicographique (ordre alphabétique basé sur le ASCII).

OCaml propose aussi des fonctions prédéfinies :
- `int_of_char 'a'` renvoie le code ASCII de `'a'` (97),
- `char_of_int 97` renvoie le caractère `'a'`,
- `String.length s` donne la longueur d'une chaîne `s`,
- `print_string s` affiche une chaîne à l'écran.

5.1 Comparateurs et expressions conditionnelles

Les opérateurs `=`, `<>`, `<`, `>`, `<=`, `>=` permettent de comparer deux valeurs. Ils sont *polymorphes*, c'est-à-dire utilisables sur différents types, à condition que les deux opérandes aient le même type.

Les conditions s'écrivent avec la syntaxe :

```
if e1 then e2 else e3
```

Exemple

```
let x = 42
let v = 10 + (if x > 0 then 3 else 4)
```

Les branches `e2` et `e3` doivent être du même type pour que l'expression soit bien typée.

6 Fonctions simples

Les fonctions se définissent avec le mot-clé `fun` :

```
fun <id> -> <expr>
```

où `<id>` est le paramètre et `<expr>` le corps.

Exemple

```
fun x -> x * x
```

Cette fonction prend un entier x et renvoie $x \times x$. Son type est noté :

```
int -> int
```

Une fonction définie avec `fun` est anonyme. On peut lui donner un nom :

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

Notation équivalente et plus concise :

```
let f x = x * x
```

Il est possible d'indiquer explicitement le type d'un paramètre :

```
let f (x:int) = x + x
```

Si l'annotation est incorrecte, OCaml produit une erreur.

Exemple

```
let f (x:int) = true
```

renvoie :

```
Error: This expression has type bool
but an expression was expected of type int
```

L'application est l'opération qui permet d'utiliser une fonction sur un argument. La syntaxe est :

```
<expr1> <expr2>
```

où `<expr1>` est une fonction et `<expr2>` son argument.

```
f 42
```

applique f à 42. On peut écrire aussi `f(42)`, mais la syntaxe usuelle reste `f 42`.

L'évaluation d'une application suit trois étapes :

1. Évaluer `<expr1>` : il doit donner une fonction de type $\tau_1 \to \tau_2$.
2. Évaluer `<expr2>` : il doit être de type τ_1.
3. Substituer la valeur de `<expr2>` dans le corps de la fonction et calculer le résultat.

Attention

Un oubli fréquent consiste à ne pas mettre de parenthèses autour de l'argument d'une fonction lorsqu'il s'agit d'une expression et non d'une simple valeur. Par exemple, si `f` est :

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

alors les deux applications suivantes ne donnent pas le même résultat :

```
let v1 = f 2 + 1
let v2 = f (2 + 1)
```

Dans `v1`, l'application de fonction est prioritaire : `f 2` vaut 4, puis on ajoute 1 $\Rightarrow v1 = 5$. Dans `v2`, les parenthèses forcent l'évaluation de `2 + 1`, donc `f 3 = 9`.

Un autre type d'erreur vient du typage. Par exemple :

```
let g x = 3.14 *. (float x)
let v = g 2 + 1
```

le compilateur renvoie :

```
Error: This expression has type float
but an expression was expected of type int
```

car la fonction `g` convertit un entier en flottant alors que l'opérateur `+` attend deux entiers.

6.1 Fonctions utiles

En OCaml, il existe plusieurs fonctions prédéfinies très pratiques pour afficher ou lire des valeurs. Elles prennent un argument (ou parfois aucun), effectuent une action et renvoient `()` du type `unit`.

Fonctions d'affichage

— `print_int : int -> unit`

```
print_int 42 (* affiche 42 *)
```

— `print_float : float -> unit`

```
print_float 3.14 (* affiche 3.14 *)
```

— `print_char : char -> unit`

```
print_char 'A' (* affiche A *)
```

— `print_string : string -> unit`

```
print_string "Hello"
```

— `print_newline : unit -> unit`

```
print_newline () (* saute une ligne *)
```

— `Printf.printf : string -> ... -> unit`

```
Printf.printf "x = %d et y = %f\n" 42 3.14
```

Fonctions de lecture

— `read_line : unit -> string`

```
let s = read_line ()
```

— `read_int : unit -> int`

```
let n = read_int ()
```

— `read_float : unit -> float`

```
let x = read_float ()
```

6.2 Fonctions sans résultats et fonctions sans arguments

En OCaml, comme en mathématiques, une fonction renvoie toujours une valeur. Mais que se passe-t-il pour une fonction comme `print_string`, qui affiche une chaîne et ne renvoie rien ?

La réponse est le type `unit`, qui ne contient qu'une seule valeur notée `()`.
Ainsi, le type de `print_string` est :

```
string -> unit
```

Il est parfois utile de définir des fonctions qui ne prennent pas d'arguments, par exemple pour

afficher un message. En OCaml, toute fonction doit avoir un argument, mais on peut utiliser le type `unit` :

```
let fname = "John"
let name = "Doe"
let message () = print_string ("Hello " ^ fname ^ " " ^ name ^ "\n")
```

Le type de `message` est :

```
unit -> unit
```

Un appel `message()` affiche :

```
Hello John Doe
```

6.3 Fonctions et portée lexicale.

Les fonctions permettent d'illustrer la notion de portée des variables.

```
let x = 42
let f y = x + y
let v1 = f 10
```

Ici, `v1 = 52`. Mais si l'on redéfinit `x` :

```
let x = 10
let v2 = f x
```

alors `v2` vaut toujours 52. La nouvelle définition de `x` masque la précédente globalement, mais la fonction `f` continue à utiliser la valeur de `x` qui était en vigueur lors de sa définition.

7 Déclarations locales

Jusqu'ici, nous avons vu deux types de variables :

- les variables globales (`let x = ...`),
- les variables paramètres de fonction (`fun x -> ...`).

OCaml permet aussi de définir des variables locales avec la construction `let ... in`.

```
let <id> = <expr1> in <expr2>
```

Exemple

```
let x = 42
let y =
  let x = 9.4 *. 3.14 in
  x +. x
let v = x + 10
```

La déclaration locale masque temporairement la variable globale `x`. Dès que l'expression locale est terminée, la portée de la variable globale est rétablie.

Comme `let-in` est une expression, elle peut être utilisée partout où une expression est attendue, par exemple dans le corps d'une fonction ou une sous-expression :

```
let v =
  let z = (let y = 9 * 9 in y + y) / 10 in
  z + z
```

8 Instructions

Une *instruction* est une opération qui modifie un état (mémoire, écran, etc.). En OCaml, toutes les instructions sont représentées par des expressions de type `unit`.

Le langage propose un opérateur de séquence `;` qui permet d'exécuter deux expressions successivement :

```
<expr1> ; <expr2>
```

La valeur finale est celle de `<expr2>`.

Exemple

```
let x = print_string "Hello\n" ; 10 + 32
```

Ici, la fonction `print_string` affiche `"Hello"`, puis l'expression `10 + 32` est évaluée, et la variable `x` reçoit la valeur `42`.

On peut enchaîner plusieurs affichages :

Exemple

```
let print_result x =
  print_string "The result is: ";
  print_int x;
  print_string " degrees celcius\n"
```

Un appel `print_result 42` affiche :

```
The result is: 42 degrees celcius
```

Blocs d'instructions.

Grâce au type `unit`, on peut écrire des conditions sans branche `else`. La condition doit être de type `bool`, et la branche `then` doit être de type `unit`.

```
if <expr> then <expr1>
```

Il faut être attentif à la priorité de l'opérateur de séquence `;`.
Dans une conditionnelle comme :

```
if <expr> then <expr1>; <expr2>
```

l'expression `<expr2>` est toujours exécutée, quelle que soit la valeur de `<expr>`. En effet, l'opérateur `;` a une priorité plus faible que la conditionnelle. Il faut donc lire :

```
(if <expr> then <expr1>) ; <expr2>
```

Pour changer la priorité, on ajoute des parenthèses :

```
if <expr> then (<expr1>; <expr2>)
```

On peut aussi utiliser un bloc `begin ... end` :

```
if <expr> then
  begin
    <expr1>;
    <expr2>
  end
```

Commentaires

En OCaml, les commentaires commencent par `(*` et se terminent par `*)`. Ils peuvent apparaître n'importe où, s'étendre sur plusieurs lignes et même être imbriqués.

```
(* ceci est
   un commentaire
   qui contient un (* autre commentaire *)
*)
```

On peut aussi insérer des commentaires au milieu d'expressions :

```
let x = (* un entier *) 42 + (* un autre entier *) 10
```

Attention

Oublier de fermer un commentaire provoque une erreur claire et précise.

```
let v = "hello"

(* ceci est
   un commentaire (* qui n'est pas *) ferme

let x = 42
```

La compilation affiche :

```
Error: Comment not terminated
```

9 Modules

Le langage OCaml met à disposition une bibliothèque standard organisée en *modules*. Un module regroupe des déclarations de types, de constantes et de fonctions pour un usage précis (listes, tableaux, entrées-sorties, etc.).
Chaque module correspond à deux fichiers partageant le même préfixe :

— le fichier d'implémentation (`.ml`) : contient les définitions,
— le fichier interface (`.mli`) : contient les signatures (types et fonctions accessibles).

Exemple

Le module `List` de la bibliothèque OCaml correspond aux fichiers `list.ml` et `list.mli`. Il définit le type des listes OCaml et de nombreuses fonctions associées.

Accès aux valeurs d'un module.

Pour utiliser une valeur définie dans un module, on écrit :

```
1 List.iter ...
```

Ici, `List` est le nom du module et `iter` le nom de la fonction.

Ouverture d'un module.

On peut aussi ouvrir un module entier avec :

```
1 open List
```

Cela rend visibles toutes ses définitions sans avoir à les préfixer. Cependant, cette pratique peut provoquer des conflits si deux modules contiennent des définitions de même nom. Par prudence, il est préférable de garder la notation préfixée.

Le compilateur OCaml charge par défaut le module `Stdlib`, qui contient de nombreuses petites fonctions de base utilisées dans tout programme.