Le langage OCaml

Julien Reichert

Ocaml

- **C**ategorical **A**bstract **M**achine + **M**eta **L**anguage. La lettre O signifie qu'on y a ajouté de la programmation objet (pas pour nous).
- Langage dit fonctionnel (essentiellement). En gros, faire un calcul, c'est évaluer une fonction. Caricature : **1** est la fonction qui ne prend pas d'argument et qui renvoie l'entier après **0**.
- En pratique, la modification de variables et l'écriture de boucles relève plutôt de la programmation dite impérative. On en fera tout de même en OCaml.
- En tout cas, c'est un tout nouveau langage qui change relativement de Python et C. Attention aux confusions !

Les données et leurs types

- OCaml est très rigoureux au niveau des types : `1` n'est pas égal à `1.0`, au point qu'il est interdit de les comparer.
- La notion de signature d'une fonction est d'autant plus importante (nom de la fonction et type des arguments et de la valeur de retour). Exemple : on écrira `somme : int list -> int`.
- Types de base : pas de dépaysement, on utilise `int`, `float` et `bool`, ainsi que `char` (caractère, qu'on entoure d'apostrophes).

Entiers, flottants et booléens

- Opérations : Comme on a dit, OCaml est très rigoureux, donc + est réservé aux **entiers**, sinon c'est `+.` et de même pour les trois autres opérations de base.

- Ainsi, / est la division euclidienne (incorrecte avec négatifs !) et `/.` la division usuelle. Reste dans la division euclidienne avec `mod`. Puissance, réservée aux flottants, avec **. Pour les booléens, les opérations sont &&, || et `not` (attention, `and` veut dire autre chose).

- Comparaisons : <=, <, >=, >, <> (ne pas utiliser !=) et = (ne pas utiliser ==).

- Conversions : `float_of_int`, `int_of_float`, entre autres.

Caractères et chaînes de caractères

- Caractères (`char`) : type de base (un seul caractère), entourés d'apostrophes.
- Chaînes de caractères (`string`) : autant de caractères qu'on veut, entourés de guillemets.
- Caractère d'indice `i` (entre `0` et `String.length(s) - 1`) de `s` : `s.[i]`. Le type de l'objet est naturellement `char`.
- Les chaînes de caractères étaient **mutables** jusqu'aux dernières versions du langage, par exemple on pouvait remplacer l'initiale : `s.[0] <- c` (`c` de type `char`, `s` non vide).

Chaînes de caractères

- Concaténer deux chaînes de caractères : `^` (infixe).
- Extraire une sous-chaîne : fonction `String.sub`, arguments : chaîne, indice de départ et longueur de la sous-chaîne à extraire, erreur en cas de débordement.
- Initialiser une chaîne : fonction `String.make`, arguments : longueur et caractère à répéter.
- Dépendances quand on déclare une chaîne comme exactement une autre (cf. `ll = l` en Python), mais vu l'immutabilité cela se voit (et dérange) moins.

Tableaux

- Similaires aux listes de Python, mais tous les éléments doivent être de même type. On aura donc des `int array`, des `string array`, des `int array array` (matrices)...
- Gestion proche de celle des chaînes : accès à un élément `t.(i)`, modification `<-`, longueur `Array.length`, sous-tableau `Array.sub`, initialisation `Array.make`.
- Fusion (d'une liste de tableaux) : `Array.concat`. Il est impossible de modifier en place un tableau pour changer sa taille.
- Écriture explicite : `[|1;2;3|]`.

Listes

- Structure neuve par rapport à celles qui ont été apprises en Python : on n'accède directement qu'à l'élément de tête, qui pointe sur le reste de la liste.
- Seulement des éléments d'un même type : `int list`, `int array list`, etc.
- Écriture explicite : `[1;2;3]`, ou alors `tete::queue` (préfixage) ou `liste@liste2` (fusion).
- Longueur : `List.length` (coût linéaire). Pas d'initialisation par `make` mais `init` ajouté récemment, pas d'extraction, pas de modification.

Nouvelles fonctions de conversion

- `int_of_string` et `string_of_int`, de même en remplaçant `int` par `float`, `bool` ou `char` ; `char_of_string` n'existe pas.
- `int_of_char` et `char_of_int` : entier = position du caractère dans la table ASCII étendue (256 caractères).
- `Array.of_list` et `Array.to_list`, coût linéaire.

n-uplets

- Structure de données de proche du pendant en Python : éléments de types quelconques, séparés par des virgules et entourés de parenthèses ou non au choix, peuvent être déconstruits.
- Mais pas de type à part entière (donc de longueur), type : produit des types des éléments (ex. `string * int list` (attention : `["",1]` est une `(string * int) list`).
- Pour des couples uniquement, fonctions `fst` et `snd`. Souvent à remplacer par une déconstruction.

Variables

- **En OCaml, la notion de variable n'existe pas !**
  ... donc quand on en parle quand même, c'est par abus.

- En pratique, on considère les références comme des variables.

- Syntaxe des affectations : `let nom = valeur;;`
  (points-virgules pour terminer une instruction composée dans une session console).
  Cela lie `nom` à la valeur pour toujours (... jusqu'à réaffectation).

Variables

- Définition locale : `let nom = valeur in expression;;`. La liaison ne se fait que dans le code en question.
- Référence en OCaml : version mutable d'une valeur immutable. Type : `'a ref`, où `'a` est le type de la valeur.
- Accéder à la valeur actuelle d'une référence `x` : déréférencer, en écrivant `!x` (erreurs de syntaxe fréquentes si espace avant oubliée).
- Modifier une référence : `x := valeur`. Cas particuliers d'une référence d'entier : `incr` / `decr`.

Exemple d'utilisation d'une référence

```
let l = ref [] in
  let i = ref 0 in
    while !i < 10 do
      print_int !i;
      l := !i::!l;
      incr i
  done; !l;;
```

Instructions (n'existent pas non plus mais on fait comme si)

- Voir les instructions comme des expressions évaluées qui ont chacune un type. Type courant (cf. Python `NoneType`) : `unit`. Valeur associée : `()`.
- Séquence d'instructions : séparées par des `;`. Type de la séquence : celui de la dernière, les autres doivent être `unit`.
- Indentation, tabulations diverses, etc. : au choix du programmeur, seule compte la lisibilité !

Disjonction de cas (if)

- Syntaxe `if condition then expr1 else expr2`. Sémantique usuelle.
- Type de `condition` : nécessairement `bool`. Possibilité d'omettre la partie avec `else`, mais alors le type de `expr1` doit être `unit`.
- Pas de `elif` ni `elseif`, il faut imbriquer (ou faire un filtrage).
- Attention : blocs sans instruction avant l'expression finale ou entourés de parenthèses.

Boucle inconditionnelle (for)

- Syntaxe : `for variable = debut to fin do bloc done` ou `for variable = debut downto fin do bloc done`, nécessairement par pas de `1` ou `-1`.
- Sémantique : bloc évalué pour les valeurs de `variable` entre `debut` et `fin` inclus, jamais si l'ordre est le mauvais.
- Type de `bloc` : nécessairement `unit`. Cf. `for i = 1 to 42 do 2+2 done;;` ne retournant rien (avec un avertissement).
- `debut` et `fin` évalués avant d'entrer : pas de perturbation. Attention : `variable` n'existe que dans la boucle.

Boucle conditionnelle (while)

- Syntaxe : `while condition do bloc done`. Sémantique intuitive.
- Type de la condition : nécessairement `bool`. Type de `bloc` : nécessairement `unit`.
- Condition évaluée avant chaque tour de la boucle.

Fonctions

- Rappel : OCaml langage fonctionnel. Notion de fonction capitale.
- Fonction : expression qui attend un (ou plusieurs, autre abus) argument (éventuellement `()` de type `unit`) et qui renvoie un objet de type bien défini.
- Signature : détermine le type des arguments et de la valeur de retour. Exemple : `fact : int -> int`
- Dans la définition d'une fonction : expression éventuellement précédée d'instructions. Valeur de retour = dernière expression.

Fonctions

- Syntaxe simple : `let nom argument(s) = expression;;` (parenthèses optionnelles, ex. : `let add x y = x + y;;`).
- Cas élaboré :

```
let fact n = let res = ref 1 in
    for i = 2 to n do res := !res * i done;
    !res;;
```

- Instruction supplémentaire utile pour les fonctions : filtrage.

Filtrage

- Deux syntaxes pour distinguer les cas, vues le plus souvent lorsque l'on définit une fonction. On ne verra qu'une de ces syntaxes.
- `match expr with` suivi de cas de filtrage suivant la valeur que prend `expr`
- Chaque cas à partir du deuxième (premier de plusieurs aussi pour l'esthétique) démarre par `|`, puis la valeur de l'objet, puis `->`, puis le résultat pour ce cas.
- Attention ! Lorsque l'on donne la valeur d'un objet, soit on utilise une constante (`0`, `true`, `[]`, etc.), soit on est en train de créer une variable locale qui masque donc d'éventuelles variables homonymes.

Cas de filtrage

- Pour l'esthétique, un cas par ligne.
- Tous les types des résultats doivent être les mêmes, ainsi que ceux des valeurs des objets aux mêmes positions.
- Le premier cas correspondant est traité et lui seul.
- Filtrage dans un filtrage : ambigu, parenthéser.

Cas de filtrage - exemples

Si deux cas ou plus donnent le même résultat, on peut omettre la partie démarrant par -> des premiers, à condition qu'aucune variable n'apparaisse (c'est un peu logique...).

Le joker _ traite le reste des cas (identique, mais moins esthétique, à l'utilisation d'un nom quelconque, si le nom ne sert pas dans la suite). Si le filtrage n'est pas exhaustif, OCaml déclenche un avertissement (et une erreur si le filtrage échoue ultérieurement).

```
let test_parite n = match n with
| 0 | 2 -> true
| k -> if k mod 2 = 0 then true else false;;
```

Cas de filtrage - exemples

Attention : le filtrage se fonde sur une reconnaissance de motifs, et pour qu'il soit optimisé certaines choses sont interdites. Comparer :

```
let test_egalite_qui_plante b1 b2 = match (b1, b2) with
| (b, b) -> true
| (_, _) -> false;;

let mauvais_test_egalite b bb = match bb with
| b -> true
| a -> false;;
```

Cas de filtrage - exemples

```
let liste_vide l = match l with
|[] -> true
|(_::_) -> false (* Ou _ -> false.
Au passage, ceci est un commentaire
et il peut porter sur plusieurs lignes. *)

let filtrage_exhaustif = function
|true -> 42
|false -> 24;;
```

Entrées et sorties

- Fonctions multiples. . . car le type est strict !
- `print_int`, `print_float`, `print_char`, `print_string`, mais pour les listes et tableaux entre autres, il faut une boucle. Valeur de retour : `unit`.
- Idem pour la lecture : `read_int`, `read_float`, `read_line`. Argument : `()`.

Entrées et sorties

- Ouverture de fichier (canal) : `open_in` (lecture) et `open_out` (écriture).
- Signatures `string` $\rightarrow$ `in_channel` et `string` $\rightarrow$ `out_channel`.
- Penser à toujours fermer ! (`close_in` et `close_out`)
- Canaux standards : `stdin`, `stdout` et `stderr`. Ne pas fermer !
- Lire et écrire dans un fichier : remplacer `read` par `input` et `print` par `output` dans les noms de fonctions, et donner en premier argument le canal.

Liaisons statiques

Comparer Python et OCaml

```
def g():
    print(1)
def f():
    g()
f()
def g():
    print(2)
f()
```

```
let g () = print_int 1;;
let f () = g ();;
f();;
let g () = print_int 2;;
f();;
```

Syntaxe supplémentaire

- Définitions simultanées : `let x = 2 and y = 2 in x + y;;`
- Contrairement au même code avec `in let`, `y` ne peut pas dépendre de `x`.
- Utilité pour des fonctions mutuellement récursives (voir chapitre suivant).

Syntaxe supplémentaire

- Fonctions locales : Notion intuitive (comme en Python).
- Fonctions anonymes (comme en Python, où on en parle peu) : `(fun x y -> x * y) 6 7;;`
- Utilité : combiner avec `List.map`, `List.fold_left`, etc.

Types construits

- Inutile : aliasage pour des types existants (`type complexe = float * float;;`).
- Type somme : définition d'un type à l'aide de constructeurs typés ou non (éventuellement celui qu'on définit).
- Exemple : `type entier = Zero | Un | Beaucoup;;`
- Plus intelligent : `type nombre = Plusinf | Moinsinf | Entier of int | Flottant of float | Fraction of int * int;;` (ressemble au type `num`)
- Éléments possibles : `Plusinf`, `Entier 3`, `Flottant(exp 1.)`, `Fraction(-4,0)` (!!!), ...

Types construits

- Type produit : se rapproche de la programmation orientée objet.
- Créer un enregistrement muni de rubriques, mutables ou non ayant chacune un type précis.
- Syntaxe :

```
type montype = {rub1 : type1 ; rub2 : type2 ;};;
let exemple = {rub1 = valeur1 ; rub2 = valeur2
;};;
```

Types construits

- Accéder à une rubrique : `exemple.rub1`.
- La modifier : `<-`, à condition que dans la définition de type on ait précédé son nom de `mutable`.
- Exemple : `type individu = {mutable nom : string; prenoms : string list; mutable age : int; sexe : bool};;`

Exceptions

- Comportements imprévus : division par zéro, accès à un élément inexistant d'un tableau, ...
  L'exécution s'arrête et l'exception est précisée par OCaml.
- Pas une exception : erreur de syntaxe ou de typage.
- Type spécifique : `exn`. Peut se substituer à n'importe quel autre type.
- Exemple : `if test then 42 else failwith "Erreur";;`

Exceptions

- Créer une exception : `exception Nom;;`, éventuellement avec un type : `exception Trouve of int;;`
- Déclencher une exception : `raise Nom;;`, et avec un type : `raise (Trouve 42);;`.
- Pour de simples messages d'erreur : `failwith "Message";;` (équivalent à `raise (Failure "Message");;`).
- Existe aussi : `invalid_arg "Message";;` (déclenche l'exception `Invalid_argument`).

Exceptions

- Rattraper une exception : `try ... with ...`.
- Première partie : une expression qui peut déclencher des exceptions.
- Deuxième partie : un filtrage sur les exceptions, qui sont alors rattrapées.
- Si le filtrage n'est pas exhaustif, une exception ne correspondant à aucun cas n'est pas rattrapée.

Exceptions

```
exception Trouve of int;;
let cherche_chaine carac s =
  try
    for i = 0 to String.length s - 1 do
      if s.[i] = carac then raise (Trouve i)
    done; -1
  with
    |Trouve ind -> ind;;
```

Modules

- La bibliothèque standard d'Ocaml est suffisamment fournie, et les modules ne sont pas aussi nombreux qu'en Python (évidemment).
- La plupart des modules classiques sont préchargés et doivent juste être ouverts, notamment `Printf`, `Random` et `Sys`. Éviter d'ouvrir `List`, `String` et `Array`.
- Un module étudié en TP doit même être chargé avant ouverture (les éditeurs que j'ai testés le font automatiquement) : `Graphics`.
- Syntaxe : `open Random;;` puis utilisation normale des fonctions ou préfixage sans ouverture : `Random.int 6;;`.

Formats

- Imprimer en une instruction une chaîne contenant des paramètres.
  Rappel : impossible d'avoir une fonction avec un nombre arbitraire d'arguments, comme `print` en Python3.

- La syntaxe proposée est héritée de C et existe aussi en Python (adapter).

- Fonctions du module `Printf` : `printf` (imprimer dans la console), `fprintf` (dans le canal de sortie en premier argument), `eprintf` (dans le canal d'erreurs) et `sprintf` (produit une chaîne).

Formats

- L'argument principal n'est pas vraiment une chaîne, mais un format contenant des marqueurs de valeurs paramétrables.
- Dans la chaîne, %d signale un entier, %f un flottant, %c un caractère, %s une chaîne... et %% le symbole %.
- Après la chaîne, les valeurs paramétrables doivent être renseignées dans l'ordre, comme si la chaîne était une fonction à plusieurs arguments.
  D'ailleurs on obtient une fonction s'il manque des valeurs !

Formats

Exemples :

```
let table_multiplication m n =
  for i = 0 to m do
    Printf.printf "%d x %d = %d\n" i n (i*n)
  done;;

let date jour mois an =
  Printf.sprintf "%02d/%02d/%d" jour mois an;;

let imprime_format canal n =
  Printf.fprintf canal "Vous avez %d messages." n;;
```