— Cours d'Ocaml —

RIPOPÉE INITIATIQUE

RYAN BOUCHOU

Sommaire

1 Introduction

OCaml est un langage de programmation qui peut être compilé et interprété. Manifestement, il est à la fois fonctionnel (ie que le calcul est considéré comme une évaluation successives de fonctions) et impératif. Il notamment porteur des caractéristiques suivantes:

→ Garbage collection (GC) pour la gestion automatique de la mémoire. A intervalle régulier, l'algorithme de gestion automatique de la mémoire recycle les parties de la mémoire préalablement allouée puis inutilisées.

→ Polymorphisme paramétrique , qui permet la construction d'abstractions qui fonctionnent sur différents types de données.

→ Bon support pour la programmation immuable , c'est-à-dire la programmation sans effectuer de mises à jour destructives des structures de données.

→ Typage inféré par le compilateur : vous n'avez donc pas besoin d'annoter chaque variable de votre programme avec son type. Au lieu de cela, les types sont déduits en fonction de la façon dont une valeur est utilisée.

→ Programmation orienté objet, et récursive.

1 La compilation à partir d'un fichier source

On se muni d'un compilateur *ocamlc*, et on rédige son code dans un ficher *source.ml*. Alors, la commande UNIX pour compiler son fichier et l'exécuter est la suivante:

Syntaxe

ocamlc source.ml - o executable
./executable

2 L'usage d'un système interactif:

En utilisant un interpréteur lancé dans le shell, l'utilisateur rédige ses expressions Ocaml sur les lignes initiées par #, qui sont compilées à la volée, exécutées, et dont la résultante s'affiche en dessous.

→ Pour accéder à l'interpréteur: *utop*

→ Au sein de l'interpréteur:

- Implémenter une définition: *definition;;*
- Importer un fichier source: *#use "source.ml";;*
- Sortir de l'interpréteur: *exit 0;;*

Les phrases Ocaml sont des expressions simples, ou bien des définitions d'identifiants par le biais du mot-clé `let`.

Exemple

```
# 1 + 2 * 3;;
- : int = 7
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312
```

2 Expression en OCaml

1 Définitions

En OCaml, les définitions ont une portée bien précise, caractérisée par la construction `let x=content in e` où la valeur de localité de `x` est l'expression `e`

1.1 Non mutables

La définition d'identifiants comme vu précédemment met en lumière trois éléments. D'abord que la **précision des types est facultative**. Ensuite, qu'**une déclaration requiert dans le même temps une affectation**. Enfin, qu'**une telle déclaration n'est pas modifiable**.

Rappel: Nuance dans l'affectation

> *The `let` binding is not an assignment, it introduces a new identifier with a new scope.*
> *Le nom d'une variable doit commencer par une minuscule*

1.2 Mutables: Les références

On peut définir une variable mutable en utilisant le mot-clé `ref`. Dès lors, on se munit de la syntaxe suivante:

Syntaxe

> *`let x = ref content` pour déclarer une variable mutable `x`*
> *`!x` pour accéder à la valeur `content`*
> *`x:=new_content` pour modifier la valeur de `x`*

2 Les fonctions

2.1 Définition usuelle

Pour définir une fonction on utilise la syntaxe suivante:

```
let nom (arg_1 : type_1) ... (arg_n : type_n) : type = expression
```

où

→ *nom* est identificateur

→ $type_i$ est le type de l'argument

→ *type* est le type de sortie de la fonction

2.2 Fonctions anonymes

En OCaml, on peut écrire une fonction comme une simple expression. C'est à dire sans lui donner de nom, et sans `let`. Pour cela, on utilise la syntaxe suivante:

```
fun (arg_1 : type_1) ... (arg_n : type_n) : type -> expression
```

2.3 Exemple

```
    # let compose f g = fun x -> f (g x);;
    val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

2.4 Fonctions récursives

Au contraire des langages impératifs, où l'appel de fonctions est coûteux (notamment spatialement, avec une accumulation des appels dans la piles ; et subséquemment, une saturation de celle-ci), les langages fonctionnels sont spécifiquement propices aux définitions récursives. Ainsi, lorsque l'on souhaite appeler une fonction dans le corps de celle-ci sur une instance différentes, il suffit d'adjoindre le mot-clé `rec` à `let`

Exemple
Amusons-nous à calculer la suite de Fibonacci, en version impérative et en version récursive.

```
let fibo_imp ( n : int ) : int =
    let a = ref 1 in
    let b = ref 1 in
    for i=0 to (n-1) do
        let temp = (!b) in
        b:=(!a)+(!b);
        a:=temp;
    done;
    (!b)
```

```
let rec fibo_rec ( n : int ) : int =
    if n=0 || n+1 then 1
    else fibo_rec (n-1) + fibo_rec
    ↪  (n-2)
```

Une programmation récursive nous épargne 2 références, et s'avère beaucoup plus aisé à implémenter (quoi qu'il ne faille pas négliger de manière générale la complexité d'un programme au profit de la rapidité de son implémentation).

2.5 Fonctions mutuellement récursives

Plusieurs fonctions sont dites mutuellement récursives lorsqu'elles s'appellent de façon croisées.

Syntaxe

```
let rec f_1 (a1 : t1 ) ... (a_n : t_n) : type_1 = e1
    and f_2 (b1 : t1 ) ... (b_n : t_n) : type_2 = e2
    ...
    and f_n (o1 : t1 ) ... (o_n : t_n) : type_n = en
```

$\forall i \in [1..n], e_i$ *est une expression qui peut faire intervenir,* $\forall j \in [1..n]$, f_j *et les* arg_j.

Exemple

```
let rec p_n ( nn : int ) : int =
(* Calcule les termes de la suites p(n) *)
if nn = 0 then 3
else 2*(q_n (nn-1) )

and q_n ( nn : int ) : int =
(* Calcule les termes de la suites q(n) *)
if nn = 0 then 4
else 7*(p_n (nn-1)) + 6

let pq ( n : int ) : int*int = ( (p_n n),(q_n n) )
(* Affiche le couple h (p(n),q(n)) *)
```

2.6 Polymorphisme fonctionnel

Il est possible de définir des types paramétrés (cf partie Polymorphisme paramétrique); dès lors, il parait tout à fait acceptable que les fonctions admettent des arguments ayant pour type des variables.

```
# let id ( x : 'a ) : 'a = x;;
val id : 'a -> 'a = <fun>
```

2.7 L'alternative

Comme son nom l'indique, elle est l'équivalent du if else en C. La syntaxe est la suivante:

```
    if c_1 then e_1
    else if c_2 then e_2
    [...]
    else e_3
```

→ Les c_i sont des expressions booléennes.

→ Les e_i sont des expressions de même type.

3 Définition de types

3.1 Types simples

→ Types prédéfinis: int / float / string / unit / char / bool

→ Variable de type: `'a` signifie que l'objet caractérisé par un tel type peut prendre n'importe quel type prédéfinis.

3.2 Définition locale

Lorsque l'on implémente des types abstraits, il peut être judicieux de renommer leurs types représentatifs, qu'ils soient paramétrés ou non, comme l'on réaliserait des `typedef` en C.

Syntaxe

```
    let identificateur in expression_de_type
```

3.3 Types produits

Il s'agit du type d'un n uplet: $(t_1 * t_2 * \cdots * t_n)$ avec $\forall i \in [1..n]$ un type possiblement différent.
Comme chaque champ n'a pas de nom, on accède à ceux-ci grâce à la syntaxe suivante:

Syntaxe

```
        let c=(a,b,...,x) in expression
```

3.4 Types fonctionnels

Si on considère la fonction `fun (x : tx) (y : ty) (z : tz ) ... : tf -> expression`
Alors, elle a pour type: $t_x \to t_y \to t_z \to ... \to t_f$

> Rappel : Une fonction à n argument associe au premier une fonction des n-1 restants.

3.5 Définir un type paramétré

Syntaxe

```
type ('a1,'a2,...,'an) nomdutype = exp_de_type
```

3.6 Exemple

```
type 'a couple = ('a)*('a)
let couple : bool couple = (true,false)
```

3.7 Définir un type somme

Syntaxe

```
type id = Constructeur_1 | ... | Constructeur_n

type id =
    | Constructeur_1
    | Constructeur_2 of exp_de_type

type ('a,'b,...,'n) id =
    | Constructeur_1 of exp_de_type
    | Constructeur_2
```

Au début de chacune des sections, séparées par des |, se trouve un constructeur. On peut les nommer comme on veut, tant que leur nom commence par une capitale.
Si un constructeur peut être utilisé pour définir une valeur, il est suivi de **of** et d'une expression de type (potentiellement paramétrée).

3.8 Type somme paramétré prédéfini

Le type 'a option est déjà défini en OCaml. Il contient le valeur None, qui signifie "pas de donnée" ou une valeur de type 'a en-capsulée dans le constructeur Some.
Ce type peut servir à stocker des données incomplètes (comme les notes d'une classe à un devoir où None est la note d'un absent) ou à gérer des fonctions partielles (on renvoie None quand la fonction n'est pas définie sur les valeurs passées en argument, comme la moyenne d'une liste vide).

Syntaxe

```
type 'a option =
    | None
    | Some of 'a
```

3.9 Type somme paramétré et récursif

Exemple d'une pile:

```
type 'a pile =
| PileVide
| PileNonVide of ('a*('a pile))
```

3.10 Types enregistrements

Les types produits, appelés aussi types enregistrement ou record en anglais, sont l'équivalent des structures en C : ils permettent de rassembler des données de types différents, comme un n-uplet, en leur donnant un nom (avantage par rapport au n-uplet : on n'a plus besoin de connaître l'ordre des données, seulement le nom des champs associés).

- Nouvelle expression de type associée On définit un type produit grâce à la syntaxe suivante:

```
type identificateur = {nom_du_champ_1 : type_1 ; ... ; nom_du_champ_n
↪  : type_n }
```

- Définition d'un type enregistrement (paramétré)

```
type ('a, ..., 'n) identificateur = {nom_du_champ_1 : type_1 ; ... ;
↪  nom_du_champ_n : type_n }
```

- Expressions associées : définir une valeur de ce nouveau type produit

```
let identificateur : nom_du_type = {nom_du_champ_1 = val1 ; ... ;
↪  nom_du_champ_n = valn }
```

- Accès à un champ d'une valeur de ce nouveau type produit

```
identificateur.nom_du_champ_n
```

- Modification d'un champ mutable

```
identificateur.nom_du_champ<-content;
```

3 Filtrage

1 Principe

Un filtrage prend la forme suivante:

```
match valeur with
    | motif    ->  résultat
    | motif    ->  résultat
    ...
```

On veillera à faire des filtrages non redondants (ce qui déclenche un warning de OCaml, comme vu en cours) et exhaustif, c'est-à-dire dont les motifs couvrent toutes les valeurs possibles du type filtré, quitte à mettre des failwith pour les valeurs non conformes aux hypothèses.
On voit qu'on peut utiliser le tiret du bas pour éviter de nommer une variable non utilisée, ou pour désigner "toutes les autres valeurs".

2 Exemples

```
let compte (x:int) : string =
match x with
    | 1 -> "un"
    | 2 -> "deux"
    | _ -> "bcp"

let compte_couple (c:int*int) : string =
match c with
    | 1,_  -> "un peu"
    | 2,_ -> "deux peu"
    | 4,3 -> "quatre trois"
    | _,3  -> "trop"
    | _ -> "bcp"

 match identificateur with
    | {nom_du_champ_1 = val_1 ; ... ; nom_du_champ_n = val_n} -> expression
    | {_;nom_du_champ_2 = val2 ; _ ;...; _} as identificateur -> expression
```

3 Combinaison de motifs

La combinaison de plusieurs motifs permet d'obtenir un nouveau motif qui pourra déstructurer une valeur selon l'un ou l'autre de ses motifs originaux. La forme syntaxique est la suivante :

$$|p_1|...|p_n$$

Elle construit un nouveau motif par combinaison des motifs p_1, ...et p_n. La seule contrainte forte est de refuser tout nommage à l'intérieur de ces motifs. Donc chacun d'eux ne devra contenir que des valeurs constantes ou le motif universel.
L'exemple suivant montre comment vérifier qu'un caractère est une voyelle.

```
# let est_une_voyelle c = match c with
 |'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
 | _ -> false
```

4 Nommage d'une valeur filtrée

Lors d'un filtrage de motif, il est parfois pratique de nommer tout ou partie du motif.
La forme syntaxique suivante introduit le mot clé as qui associe un nom à un motif.

```
identificateur as new_id
```

Ceci est utile lorsqu'on a besoin de déstructurer une valeur tout en la conservant dans son intégralité. Dans l'exemple suivant, la fonction min-rat rend le plus petit rationnel d'un couple de rationnels. Ces derniers sont représentés par un couple numérateur et dénominateur.

```
let min_rat cr = match cr with
|((_,0),c2) ->  c2
| (c1,(_,0)) ->  c1
| (((n1,d1) as r1), ((n2,d2) as r2)) ->
     if (n1 *  d2 ) < (n2 * d1) then r1 else r2
```

Pour comparer deux rationnels, il est nécessaire de les déstructurer pour pouvoir nommer leur numérateur et leur dénominateur (n1, n2, d1 et d2), mais il faut rendre le couple initial (r1 ou r2). La construction as nous permet ce nommage de parties d'une même valeur. Cela évite de devoir reconstruire le rationnel retourné en résultat.

4 Les Modules

1 Module List

1.1 Le type

Il existe un module prédéfini en OCaml qui implémente le type abstrait des listes chaînées. ⚠Cette structure est non mutable. On présente l'implémentation de son type.

Syntaxe

```
type 'a t = 'a list =
|[]                       (*représente la liste vide*)
|(::) of 'a * 'a list     (*constructeur "cons" qui permet de joindre un
↪  élément, à sa gauche, ainsi qu'une liste (éventuellement la liste
↪  vide) à sa droite. Il crée une nouvelle liste pour représenter la
↪  résultante de cette opération.*)
```

1.2 Méthodes

On peut utiliser un certains nombres de méthodes dont on obtiendra une liste exhaustive sur la documentation. On utilise ces méthodes avec la syntaxe `List.methode ...`

→ `length : 'a list -> int`

→ `hd : 'a list -> 'a` Return the first element of the given list.

→ `rev : 'a list -> 'a list`

→ `init : int -> (int -> 'a) -> 'a list`
init len f is [f 0; f 1; ...; f (len-1)], evaluated left to right.

→ `rev_append : 'a list -> 'a list -> 'a list` Renverse l1 dans l2

→ `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
fold_left f init [b1; ...; bn] is f (... (f (f init b1) b2) ...) bn.

1.3 Filtrage

```
let f ( l : 'a list )  : 'a =
    match l with
    | [] -> ...
    | x :: ll -> ...

let f2 ( l1 : 'a list ) ( l2 : 'b list ) =
    match l1, l2 with
    | [], []        -> ...
    | [], _         -> ...
    | _, []         -> ...
    | x::q, y::p    -> ...
```

2 Array

Il existe un modèle prédéfini en OCaml pour implémenter les tableaux, qui sont des structures abstraites statiques et cette fois-ci mutables. Il y a nécessairement homogénéité des éléments, et l'accès à ceux-ci est en O(1).

2.1 Principe

Les tableaux sont indicés par $[1..n]$, n étant la taille de celui-ci.

Syntaxe

`[||]` est le tableau vide
`;` est le sépérateur des éléments

Syntaxe

`t.(i)` accès
`t.(i)<-content` affectation

2.2 Méthodes

On peut utiliser un certains nombres de méthodes dont on obtiendra une liste exhaustive sur la documentation. On utilise ces méthodes avec la syntaxe `Array.methode ...`

→ `make : int -> 'a -> 'a array`

→ `length : 'a array -> int`

→ `init : int -> (int -> 'a) -> 'a array`

→ `copy : 'a array -> 'a array`

2.3 Matrices

⚠La méthode qui consiste à utiliser `make : int -> 'a -> 'a array` avec une nouvelle commande Array.make en facteur d'initialisation ne produit pas l'effet attendu. Et pour cause, toutes les cases du tableau vont pointer vers le tableau d'initialisation en réalité unique, et stocké à une et une seule adresse. En conséquence de quoi, nous n'aurons fait que le passer par valeur dans chacune des cases. Pour remédier à cela, il faudrait initialiser chacune des cases du tableau "principal" en réalisant un Array.make (à l'aide d'un for). Sinon, on peut utiliser le module déjà existant:

Syntaxe

`make_matrix : int -> int -> 'a -> 'a array array`

2.4 Chaînes de caractères

On accède aux éléments d'une chaines de caractères `let chaine = "coucou hibou"` comme suit: `chaine.[i]` si tant est que i soit un indice valide bien évidemment.
Les chaines de caractères, dans les dernières versions d'OCaml, ne sont plus des objets mutables ; et subséquemment, il faut utiliser le module **Bytes** pour les manipuler sous forme modifiable.

On donne les méthodes suivantes:

→ `of_string : string -> bytes`

→ `to_string : bytes -> string`

→ `get : bytes -> int -> char`
get s n returns the byte at index n in argument s

→ `set : bytes -> int -> char -> unit`
set s n c modifies s in place, replacing the byte at index n with c.

3 Tables de hachage

On utilise le module **Hashtbl**

3.1 Principe

En cours d'écriture...

3.2 Méthodes

→ `create : ?random:bool -> int -> ('a, 'b) t`

→ `add : ('a, 'b) t -> 'a -> 'b -> unit`

→ `find : ('a, 'b) t -> 'a -> 'b`
Hashtbl.find tbl x retourne la valeur associée à la clé x in tbl, ou raises Not_found si aucun enregistrement avec cette clé n'est présent dans tbl.

→ `mem : ('a, 'b) t -> 'a -> bool`
Hashtbl.mem tbl x retourne si la clé x est présente dans tbl.

5 Programmation orienté objet

En cours d'écriture...