Programmation avec OCaml

1 Variables

Pour créer une variable globale, écrire `let nom_variable = valeur_variable;;`.

Pour créer une variable locale, écrire `let nom_variable = valeur_variable in expr;;`. La variable existe alors uniquement dans l'expression `expr`.

Remarque 1.1. Une définition globale est toujours faite le plus à gauche possible, en dehors de toute fonction ou expression. Une définition locale peut être faite au milieu d'une expression ou fonction.

```
let x = 3;; (*x globale*)

let y = 5 in y+5;; (*y locale*)

let f x = let y = 5 in 3*y+x;; (*y définie dans une fonction*)
```

Les variables sont **immuables**, on ne peut pas changer leur valeur.

2 Types

Ocaml est *fortement* typé : à la compilation ou l'interprétation, les types doivent correspondre entre eux sous peine d'une erreur. Ocaml n'admet pas de conversion implicites (cast) de types.

Ocaml détermine automatiquement le typage des expressions via un algorithme d'inférence.

2.1 Entiers

Définition 2.1. `int` est le type des entiers relatifs pour Ocaml.

Les entiers de Ocaml vont de -2^{62} à $2^{62}-1$ et tous les calculs sont effectués modulo 2^{62}.

Les opérations possibles sont `*`, `+`, `-`, `/`, `mod`, `abs`.

Remarque 2.1.

- `a/b` est le quotient de la division euclidienne de `a` par `b` si `a` et `b` sont positifs. Si le quotient est négatif et pas un entier, l'opération renvoie sa partie entière plus 1 (par exemple $a=-5$ et $b=2$ renvoie -2).
- `a mod b` est un reste de la division euclidienne de `a` par `b` qui vérifie `a = (a/b)*b + (a mod b)` quels que soient les signes de `a` et `b`.

2.2 Flottants

Définition 2.2. `float` est le type des nombres flottants (nombres à virgule flottante) pour Ocaml.

Les opérations possibles sont `*.`, `+.`, `-.`, `/.`, `**` et les fonctions usuelles : `sqrt`, `abs_float`, `floor`, `ceil`, `exp`, `log` (logarithme néperien), `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, ...

Remarque 2.2. Caml ne fait pas la conversion des flottants vers les entiers. Pourtant, il existe un lien naturel entre `4` et `4.`. Si on veut traduire un entier en un flottant (ou l'inverse), on utilise les coercitions (on «force la main» à Caml) suivantes :

- `float_of_int : int -> float` (on peut également utiliser `float : int -> float`)
- `int_of_float : float -> int` qui effectue l'arrondi vers 0 : `int_of_float 2.6;;` renvoie `2` et `int_of_float (-2.6);;` renvoie `-2`

2.3 Booléens

Définition 2.3. Les booléens sont représentés par le type `bool`. Les variables booléennes ne prennent que deux valeurs `true` et `false`.

Les opérateurs associés sont `not` pour le non logique, `&&` pour le et logique et `||` pour le ou logique (non-exclusif).

Remarque 2.3. Les opérateurs `&&` pour le et logique et `||` sont paresseux ; c'est à dire que dans les expressions `&&` pour le et logique et `||`, `e2` sera évalué uniquement si la valeur de `e1` ne suffit pas à connaître le résultat.

Définition 2.4. Les booléens permettent de construire et d'évaluer des conditions en utilisant les opérateurs de comparaison : **<**, **>**, **>=**, **<=**, **=**, **<>**.

La comparaison fonctionne sur tous les types, mais on ne peut comparer que des objets du le même type.

2.4 Caractères et chaînes de caractères

Définition 2.5. Les caractères sont de type **char** et les chaînes de caractères de type **string**. Les caractères utilisent des apostrophes **'z'** et les chaines de caractères des guillemets **"z"** ou **"zouave"**.

Une chaîne est comme un tableau de caractères. Une chaîne n'est pas *mutable*. On peut récupérer le n-ieme caractère avec la commande **String.get**

```
let s = "Bonjour";;
String.get s 1;; (*Cette expression vaut 'o'*)
```

On peut récupérer la longueur d'une chaîne de caractère avec la fonction **String.length**.

```
String.length s;; (*Cette expression vaut 7*)
```

On peut concaténer des chaines de caractères avec l'opérateur. Par exemple `"hello "^"world";;` donne `"hello world"`.

Il est possible d'obtenir le code ASCII d'un caractère et réciproquement en utilisant les fonctions **int_of_char** et **char_of_int**.

On peut réaliser des affichages avec **Printf.printf**, qui prend les mêmes arguments qu'en C, sans parenthèses.

```
let x = 5;;
Printf.printf "%d" x;;
```

2.5 Type tuples

On peut créer des tuples en Ocaml. C'est utile pour stocker plusieurs données ensemble de manière non structurée. Les éléments d'un tuple peuvent ne pas être du même type.

```
let t = (5,4.5,'a');; (*Création*)
let (a,b,c) = t in a+3;; (*Séparation des éléments par filtrage*)

let t' = (5, 6, (1,2));;
let (a,b,(c,d)) = t' in a+b+c+d;; (*Séparation d'une paire dans un tuple*)
```

Le type d'un tuple est le produit des types de tous les éléments qui le compose. Par exemple **t** est de type **int*float*char** et **t'** de type **int*int*(int*int)**.

2.6 Type fonction

Une fonction en Ocaml qui a un **int** associe un **int** sera de type **int->int**. La flèche indique l'application de fonction.

Si la fonction a plusieurs arguments, par exemple $f(x,y) = x + y/2$, le type contient plusieurs flèches : ici **float -> float -> float** .

2.7 Type polymorphe

Tout a un type en Ocaml mais, parfois, ce type n'est pas précisément défini.

Par exemple, l'opérateurs de comparaison **=** (et les autres) permettent de comparer des entiers ou des flottants ou des chaînes de caractères, etc... Cet opérateur est de type **'a -> 'a -> bool**.

On dit que son type est polymorphe. Le polymorphisme est indiqué par l'apparition de variables de type ici **'a**. (mais ça peut aussi être **'b**, **'c**, etc...)

'a désigne type quelconque pour le moment, mais qui sera déterminé lorsqu'on appliquera la fonction.

Remarque 2.4. La syntaxe **(variable:type)** existe pour donner des indications de type, "La plupart du temps elle est inutile et redondante, mais elle peut servir pour de la disambiguation, pour débugger certaines erreurs de type [...]" (Manuel de Ocaml)

Exemple 2.1.

```
let f x y = x<y;; (*Type 'a -> 'a -> bool*)
let f' (x:int) (y:int) = x<y;; (*Type int -> int -> bool*)
```

2.8 Le type `option`

Définition 2.6. Le type **`option`** est défini ainsi : **`type`** `'a option = None | Some` **`of`** `'a`.
Lorqu'on l'utilise, **`None`** désigne l'absence d'information, et **`Some x`** désigne la présence d'une information x.

3 Conditionnelle

Définition 3.1. Les expressions conditionnelles sont de la forme : **`if e1 then e2 else e3 ;;`** dans laquelle **`e1`** doit être de type **`bool`**, **`e2`** et **`e3`** peuvent être de n'importe quel type mais doivent avoir le même.

```
let y = if x = 0 then 3
                 else 5;;
```

4 Fonctions

4.1 Définition des fonctions

Pour définir une fonction d'une variable, deux méthodes :

- Avec **`fun`**, **`let`** `f =` **`fun`** `x -> expression;;`
- Sans **`fun`**, **`let`** `f x = expression;;`, **sans parenthèses.**

Pour utiliser la fonction, on ne met pas de parenthèses. **`let`** `z = f 34;;`
Pour définir une fonction de plusieurs variables, deux méthodes (qui ne donnent pas le même type) :

- Avec un tuple, **`let`** `f (x,y) = x+y;;`
- Par curryfication, **`let`** `f x y = x+y;;`

La curryfication permet l'application partielle des fonctions, uniquement sur la variable la plus à gauche.
Par exemple si on définit **`let`** `f x1 x2 x3 = x1+x2+x3;;`, **`f 6`** est la fonction **`f`** avec $x_1 = 6$. C'est une fonction à 2 variables x_2 et x_3.

4.2 unit

Une fonction qui ne renvoie rien est de type de retour **`unit`**.
Le type **`unit`** contient une unique valeur notée **`()`**.
Une fonction qui ne prend pas d'arguments a en fait un argument de type **`unit`**. Par exemple **`let`** `g () = 2;;`. Pour l'appeler on fait **`let`** `z = g ();;`

4.3 Filtrage

Le filtrage par motif (*pattern matching*) est une syntaxe permettant d'effectuer des disjonctions de cas sur la forme des expressions.

Définition 4.1. Un filtrage a la syntaxe suivante :

```
match expr with
  | filtre_1 -> valeur_1
  .
  .
  .
  | filtre_n -> valeur_n
```

`expr` est une expression d'un certain type. On peut matcher une variable (**`match x with..`**), un n-uplet de variables (**`match x1, x2, x3, x4 with ...`**), le résultat d'une fonction, ...
Les filtres sont des expressions du même type que **`expr`** qui peuvent être :

- des constantes de type **`int`**, **`float`**, **`char`**, ... :

```
match x with
  | 0 -> "zéro"
  | 1 -> "un"
```

- des variables : les variables utilisées sont locales et n'existent que dans la ligne commençant par `|`

```
match x with
  | 0 -> -1
  | y -> 2*y
```

- `_` pour indiquer que tout cas ne correspondant pas aux filtres précédents doit être accepté. Ce filtre doit toujours être le dernier.

```
let x= 2 in match x with
  | 2 -> 6
  | _ -> 3;;
```

renvoie `- : int 6`

```
let x= 5 in match x with
  | 2 -> 6
  | _ -> 3;;
```

renvoie `- : int 3`

```
let x= 2 in match x with
  | _ -> 3
  | 2 -> 6;;
```

renvoie `Warning : this match case is unused.`

- Un constructeur de types. Voir d'autres exemples plus tard.

```
let t = (3,(3,5),9) in match t with
 |(_,_,(a,0)) -> a+1
 |(_,_,(a,_)) -> a;;
```

Remarque 4.1. Le compilateur émettra systématiquement un avertissement (*warning*) si le filtrage n'est pas exhaustif.

Dans le cas où le filtrage est volontairement non-exhaustif, on peut utiliser `failwith` qui crée une erreur à l'exécution.

Par exemple, pour une fonction où tous les autres cas que 0 ou 1 n'ont pas de sens :

```
match x with
  | 0 -> 1
  | 1 -> 0
  | _ -> failwith "l'entrée n'est pas binaire"
```

4.4 Filtrage avec condition

On peut ajouter une garde à un filtre ; c'est à dire filtrer selon une condition avec la syntaxe suivante : `| filtre when condition -> valeur` où `condition` est une expression booléenne.

Remarque 4.2. La construction :

```
match expr with
  | filtre when condition -> valeur 1
  | filtre when not condition -> valeur 2
```

est équivalente à

```
match expr with
  | filtre -> if condition then valeur 1 else valeur 2
```

5 Aspects fonctionnels, récursivité

5.1 Récursivité

Définition 5.1. Une fonction récursive est une fonction qui fait appel à elle-même.

Pour qu'une fonction soit récursive en Ocaml, il faut utiliser le mot-clé `rec` dans sa définition : `let rec f = ...`

5.2 Récursion mutuelle

Définition 5.2. Le mot clé `and` permet de définir mutuellement deux fonctions ou deux types.

Exemple 5.1. Une définition récursive de deux fonctions `pair` et `impair` qui déterminent si un entier est pair ou impair.

```
let rec pair n = (n=0) || impair (n-1)
  and impair n = (n<>0) && even (n-1) ;;
```

5.3 Ordre supérieur

Les fonctions sont des variables comme les autres.

- Une fonction peut être un argument d'une fonction.
- Une fonction peut être le résultat d'une fonction.
- Un enregistrement peut avoir une fonction comme valeur pour un de ses champs.

```
let f g x = g (x+5);;
```

6 Données structurées

6.1 Enregistrements

C'est l'équivalent des **struct** C. Attention, la syntaxe est similaire mais pas la même.

Définition 6.1. Un type enregistrement est défini par une suite de paires **champ : type** séparées par des points-virgules : **type** nom_type = **{**champ_1 : type_1 ; ... ; champ_n : type_n**}**.

Les noms des champs doivent commencer par une minuscule.

Exemple 6.1. Par exemple, pour les nombres complexes, on définit

```
# type complexe = { re:float ; im:float};;
```

Pour créer un enregistrement, on dispose de deux méthodes :

- donner la valeur de tous les champs :

```
let x = { champ_1 = valeur_1; ... ; champ_n = valeur_n};;
```

- utiliser un autre enregistrement du même type en changeant certains champs :

```
let y = { x with champ_i1 = expr_i1; ... ; champ_in = valeur_in};;
```

Exemple 6.2. qu'on utilise de la manière suivante :

```
# let z = { re=1.6 ; im = -2.3} ;;
val z : complexe = {re = 1.6; im = -2.3}
```

Définition 6.2. Pour accéder aux valeurs stockées dans un enregistrement, on a deux méthodes :

- la notation pointée **expr.champ** comme **let** partie_relle = z.re;;
- un filtre comme

```
# let { im = x; re = y} = z ;;
val y : float = 1.6
val x : float = -2.3
```

6.2 Énumérations

Les énumérations sont des types qui ne peuvent prendre qu'un nombre fini de valeurs.

```
type couleur = Pique | Coeur | Carreau | Trefle;;
```

Les noms des valeurs possibles doivent avoir une initiale en majuscule. **Pique | Coeur | Carreau | Trefle** sont considérées comme des constantes, on les nomme *constructeurs*.

Les valeurs d'une énumération sont ordonnées de gauche à droite : **Pique < Coeur < Carreau < Trefle**.

Définition 6.3. Les constructeurs peuvent avoir des arguments. Par exemple, pour le type **hauteur** :

```
type hauteur = Roi | Reine | Valet | Point of int;;
```

les constructeurs **Roi**, **Reine**, **Valet** sont constants et sont directement les valeurs. Le constructeur **Point** a un argument de type **int**.

Pour construire un élément de type **hauteur**, on applique un des constructeurs : **let** c1 = Point 5 ou **let** c2 = Roi

Exemple 6.3.

```
type carte = { c : couleur; h : hauteur; };;
let roi_de_pique =  { c = Pique ; h = Roi} ;;
```

6.2.1 Type récursif

Le principe de la définition récursive s'applique aussi aux types énumération. Pour un type **t** de données récursif :

- on définit des valeurs sans récursion (au moins une). Ce sont les valeurs de base.
- on définit comment une valeur de type **t** se construit à partir d'une valeur de type **t** (et d'autres arguments si nécessaire).

Exemple 6.4. La liste est un type récursif :

```
type ilist = Vide|Element of int*ilist;;

let l = Element(1,Element(2,Element(3,Vide)));;
```

6.3 Listes

Définition 6.4. Le type **list** est prédéfini en Ocaml et permet de représenter des listes de valeurs qui ont toutes le même type. Le type d'une liste qui contient des entiers est **int list**. Si elle contient un type **'a**, alors c'est **'a list**.

Pour définir une liste, on peut

- utiliser des crochets :

```
# let l = [] ;;  (* liste vide *)
val l : 'a list = []
# let l = [ 1; 5; 6; 11] ;;  (* liste d'entiers *)
val l : int list = [ 1; 5; 6; 11]
```

- ajouter les éléments en tête d'une liste existante : **t::q** où **t** est un élément de même type que les éléments de la liste **q**.

```
let k = 3::[5];; (*k est [3;5]*)
```

Définition 6.5. Pour récupérer les éléments d'une liste, on utilise un filtrage. On ne peut récupérer que le premier élément, il faut itérer avec une fonction récursive pour récupérer les autres.

```
let premier_element l = match l with
                        | [] -> failwith "liste vide"
                        | x :: l' -> x;;
```

Définition 6.6. L'opérateur de concaténation renvoie une liste composée des deux listes en argument :

```
let l1 = [ 1; 2; 3] @ [ 4 ; 5] ;; (*l1 est [1; 2; 3; 4; 5]*)
```

La fonction **List.length** renvoie le nombre d'éléments d'une liste :

```
# List.length l1 ;; (*Donne 5*)
```

Les fonctions **List.hd** et **List.tl** renvoient la tête et la queue de la liste :

```
# List.hd l1;; (*Donne 1*)
```