

LIFPF – Programmation fonctionnelle

CM2 – OCaml

Licence informatique UCBL – Printemps 2022–2023

<https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md>

OCaml - caractéristiques

- Langage fonctionnel
- Fortement typé
 - Inférence de type
- Filtrage de motifs et types inductifs
- Traits impératifs et objet

Approche de la programmation

- Expressions et valeurs
- **Pas de changement** de valeur d'une variable
- Structures inductives (CM3)
- Ordre supérieur (CM4)

Types de base

- `int` : 0, 1, -5, etc
- `float` : 0.0, 3.5, etc
- `bool` : `true`, `false`
- `char` : `'a'`, `'Z'`, `'\n'`, etc
- `string` : `"OCaml"`, `"Salut !"`, etc

Quelques opérateurs

- `int` : `+` `-` `*` `/` `mod`
- `float` : `+` `.` `-` `.` `*` `.` `/` `.`
- `bool` : `&&` `||`
- `string` : `^` `get`
- Comparaisons : `=` `!=` `<` `<=` `>` `>=`

Pas de surcharge

Chaque opérateur ne peut être utilisé qu'avec **un type**,
sauf les opérateurs de **comparaison**

Variables

Variables globales

```
let ma_variable = expression;;
```

La portée de *ma_variable* est le reste du fichier.

Variables locales

```
let ma_variable = expression_1  
in expression_2
```

La portée de *ma_variable* est *expression_2*.

La construction `let ... in ...` est elle-même une expression.

- Une fois définie, *ma_variable* ne change plus de valeur.
- Une variable non utilisée peut être nommée `_`.
On peut utiliser ce nom autant de fois qu'on veut (utile plus tard).

Variables - exemples

démo

Appels de fonction

Syntaxe

f arg1 arg2 arg3 ...

- Les arguments sont séparés par des espaces
- Les parenthèses ne sont utiles que si un des arguments est une expression complexe (par exemple au autre appel de fonction)
- les appels de fonction sont prioritaires v.-à-v. des opérateurs

demo

Définition de fonction : `let global`

Syntaxe avec `let global`

```
let ma_fonction (arg1: t1) (arg2: t2) ... : tr =  
  expr_resultat;;
```

- les *t1*, *t2*, etc sont les types de arguments
- *tr* est le type du résultat
- OCaml peut inférer automatiquement ces types

Définition de fonction : let local

Syntaxe avec let ... in ...

```
let ma_fonction (arg1: t1) (arg2: t2) ... : tr =  
    expr_resultat  
in expression2
```

- le portée de *ma_fonction* est **uniquement** *expression2*

Définition de fonction

démo

Types sommes : cas simple des énumérations

Type dont les valeurs sont prises dans un ensemble fini de possibilités

Syntaxe

```
type mon_type =  
  | Cons1  
  | Cons2  
  ...  
;;
```

- Les *Cons1*, etc sont les *constructeurs*
- Les constructeurs fabriquent les valeurs

demo

Filtrage de motifs

Syntaxe : `match ... with ...`

```
match expr with
| ConsA -> exprA
| ConsB -> exprB
...
| ma_variable -> exprVar
```

- *expr* est évaluée
- la valeur obtenue est comparée à chaque cas **dans l'ordre**
- l'expression correspondant au cas est évaluée pour obtenir le résultat
- une variable :
 - est déclarée
 - correspond à n'importe quelle valeur
 - peut être utilisée dans *exprVar*

Filtrage de motif : démo

Démo

N-uplets

Syntaxe

```
(expr1, expr2, ...)
```

Type

```
t1 * t2 * ...
```

Pattern matching

```
match expr1 with  
| (x, y, ...) -> expr2
```

Démo

Filtrage de motif sur les types de base

Il est possible d'utiliser le filtrage de motif sur les types de base
(`int`, `float`, `bool`, `char`, `string`)

Démo

Types somme avec données

- Il est possible d'associer des données à un constructeur
- Plusieurs données \rightarrow n-uplet

Syntaxe type

```
type mon_type =
  | ConsA                                (* Pas de donnée associée *)
  | ConsB of tB                          (* 1 donnée *)
  | ConsC of tc1 * tc2 * ...             (* k données *)
;;
```

Syntaxe valeurs

```
ConsA                                (* Pas de donnée associée *)
ConsB valB                            (* 1 donnée *)
ConsC (valC1, valC2, ...)             (* k données *)
```

Pattern matching avec données

- Syntaxe d'un cas : issue de la syntaxe des valeurs
- Une valeur peut être remplacée par une variable
 - La variable est initialisée avec la donnée correspondante dans la valeur de l'expression

Syntaxe

```
match expr with  
| ConsA -> exprA  
| ConsB x -> exprB  
| ConsC (x,y) -> exprC
```

Démo

Pattern matching en profondeur

Il est possible de spécifier des cas sur les données d'un constructeur.

Démo

Rappels récursivité

- Une fonction récursive est utilisée dans sa propre définition (appel *récurif*)
- En général plusieurs cas en fonction de la valeur de l'argument :
 - Au moins un cas de base (pas d'appel récursif pour ces valeurs)
 - Au moins un cas où on doit faire un appel récursif.
 - L'appel se fait sur un argument "plus petit", c.-à-d. plus proche d'un cas de base
 - Par exemple sur un entier plus petit si le cas de base est 1 ou 0
- Le commentaire décrivant la fonction est particulièrement important pour l'écriture des fonctions récursives.

Récurtivité en OCaml

- On doit placer le mot-clé `rec` entre le `let` et le nom de la fonction

Démo

Listes en OCaml

- Type prédéfini
 - `[]` représente la liste vide
 - `val :: liste` représente la liste dont la tête est `val` et le reste est `liste`.
 - On peut voir `::` comme un constructeur avec une syntaxe d'opérateur.
- Le type des éléments est unique au sein d'une liste
 - Impossible de mélanger des `int` et des `string` directement dans une liste (on peut utiliser un type somme à la place)
 - Le type des listes est `t_elt list` où `t_elt` est le type des éléments de la liste
- `::` est associatif à droite, c.-à-d.

$$x :: y :: z :: [] = x :: (y :: (z :: []))$$
- On peut écrire `[elt1; elt2; elt3]`
au lieu de `elt1 :: elt2 :: elt3 :: []`

Listes en OCaml - suite

- Le `match` s'utilise de manière naturelle sur les listes
 - particulièrement utile pour les fonctions récursives sur les listes
 - Le filtrage en profondeur permet de traiter naturellement le cas des liste de taille 1, 2, etc
- Il est possible de faire des listes de listes du moment que les listes sont homogènes
- La comparaison sur les liste est structurelle : on compare le contenu

Démo