Programmation fonctionnelle

L2 Info et Math-info, 2018–19

Marc Zeitoun

14 septembre 2018

Plan

Points importants

- Travail personnel : 2 à 3h/semaine en plus des enseignments.
- Ne pas apprendre par cœur des solutions sans les comprendre.
- Réfléchir **sur papier** à :
  - La **correction** de vos programmes.
  - L'**efficacité** de vos programmes.
- **Tester** vos programmes.

Points importants

- Travail personnel : 2 à 3h/semaine en plus des enseignments.
- Ne pas apprendre par cœur des solutions sans les comprendre.
- Réfléchir **sur papier** à :
  - La **correction** de vos programmes.
  - L'**efficacité** de vos programmes.
- **Tester** vos programmes.

Ne pas écrire une fonction sans savoir ce qu'elle doit faire

Points importants

- Travail personnel : 2 à 3h/semaine en plus des enseignments.
- Ne pas apprendre par cœur des solutions sans les comprendre.
- Réfléchir **sur papier** à :
  - La **correction** de vos programmes.
  - L'**efficacité** de vos programmes.
- **Tester** vos programmes.

Ne pas écrire une fonction sans savoir ce qu'elle doit faire

Ne pas écrire de code
avant d'avoir écrit l'algorithme en français

Plan

Moodle

- Cours en ligne : rechercher **Programmation Fonctionnelle**.
- Y accéder nécessite une **connexion** à Moodle.

Étudiant.e.s non inscrits :

- mot de passe **oK@ml !**
- m'envoyer un mail après l'inscription.

Plan du second cours

- Expressions et liaisons
- Types
- `let`
- `let ... in`
- Fonctions
- Inférence de types
- Polymorphisme
- Récursivité.

Plan

Expressions

- **Expression** : quelque chose qui se **calcule**. A

Expressions

- **Expression** : quelque chose qui se **calcule**. A
  - Un **type**
  - Une **valeur**

Expressions

- **Expression** : quelque chose qui se **calcule**. A
  - Un **type**
  - Une **valeur**
- Exemples :
  - `12 + 30`
  - `12.5 + 29.5`
  - `"4" ^ "2"`
  - `True && 4 = 2`

Quelles sont les valeurs et les types de ces expressions ?
(session OCaml sous Emacs)

Plan

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`,

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`,

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`,

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`, `+.`, `-.`, `*.`, `/.`, `**`, fonctions `sqrt`, `log`, `exp`, fonctions trigo

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`, `+.`, `-.`, `*.`, `/.`, `**`, fonctions `sqrt`, `log`, `exp`, fonctions trigo
- Fonctions de conversion entre `int` et `float`.
- `char`

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`, `+.`, `-.`, `*.`, `/.`, `**`, fonctions `sqrt`, `log`, `exp`, fonctions trigo
- Fonctions de conversion entre `int` et `float`.
- `char`
  `'a'`, `'A'`,

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`, `+.`, `-.`, `*.`, `/.`, `**`, fonctions `sqrt`, `log`, `exp`, fonctions trigo
- Fonctions de conversion entre `int` et `float`.
- `char`
  `'a'`, `'A'`, `Char.code`, `Char.chr`.

Types

Type = ensemble de valeurs et opérations sur ces valeurs.

- `bool`
  `false`, `true`, `&&`, `||`, `not`.
- `int`
  `42`, `-200`, `+`, `-`, `*`, `/`, `mod`.
- `float`
  `42.`, `420e-1`, `+.`, `-.`, `*.`, `/.`, `**`, fonctions `sqrt`, `log`, `exp`, fonctions trigo
- Fonctions de conversion entre `int` et `float`.
- `char`
  `'a'`, `'A'`, `Char.code`, `Char.chr`.
- `string`
  `"a"`, `"abc"`, `^`, `String.length`, `String.get`, `String.sub`

Définition de fonction

- En OCaml, les fonctions **sont des valeurs**.
- On définit une fonction par

```
fun x y z ... -> e
```

- la fonction n'a pas de nom (intérêt ?).

Définition de fonction

- En OCaml, les fonctions **sont des valeurs**.
- On définit une fonction par

```
fun x y z ... -> e
```

- la fonction n'a pas de nom (intérêt ?).
- `x`, `y`, `z`, ... sont les **paramètres** de la fonction.
- `e` est une expression qui est la **valeur retour** de la fonction.

Fonctions : exemples

Sous Emacs

Appel de fonction

- Il est souvent utile de nommer nos fonctions.

```
let mon_polynome = fun x -> x*x + 3*x + 1
```

Appel de fonction

- Il est souvent utile de nommer nos fonctions.

```
let mon_polynome = fun x -> x*x + 3*x + 1
```

- L'application de fonction se réalise en faisant suivre le nom de la fonction par ses arguments, **sans virgule entre eux !**

```
let y = mon_polynome 42
```

Appel de fonction (2)

- Il est souvent utile de nommer nos fonctions.

```
let f = fun x y z -> x*y*z = x+y+z
```

Appel de fonction (2)

- Il est souvent utile de nommer nos fonctions.

```
let f = fun x y z -> x*y*z = x+y+z
```

- L'application de fonction se réalise en faisant suivre le nom de la fonction par ses arguments, **sans virgule entre eux !**

```
let a = f 1 2 3
let b = f 1 (-1) 0
let c = f 10 20 30
```

Appel de fonction (2)

- Il est souvent utile de nommer nos fonctions.

```
let f = fun x y z -> x*y*z = x+y+z
```

- L'application de fonction se réalise en faisant suivre le nom de la fonction par ses arguments, **sans virgule entre eux !**

```
let a = f 1 2 3
let b = f 1 (-1) 0
let c = f 10 20 30
```

L'application est **plus prioritaire** que les autres constructions.

Appel de fonction (2)

- Il est souvent utile de nommer nos fonctions.

```
let f = fun x y z -> x*y*z = x+y+z
```

- L'application de fonction se réalise en faisant suivre le nom de la fonction par ses arguments, **sans virgule entre eux !**

```
let a = f 1 2 3
let b = f 1 (-1) 0
let c = f 10 20 30
```

L'application est **plus prioritaire** que les autres constructions.
Une fonction peut être appliquée **partiellement** ($\hookrightarrow$ Emacs).

Inférence et vérification de types

- Il **infère** (calcule) le type de chaque valeur demandée.
- Il indique les **erreurs**.

Lire chaque diagnostic !

Inférence et vérification de types

- Il **infère** (calcule) le type de chaque valeur demandée.
- Il indique les **erreurs**.

Lire chaque diagnostic !

- Exemple : on ne peut pas concaténer par `^` deux entiers.

Inférence et vérification de types

- Il **infère** (calcule) le type de chaque valeur demandée.
- Il indique les **erreurs**.

Lire chaque diagnostic !

- Exemple : on ne peut pas concaténer par `^` deux entiers.
- OCaml **infère** aussi le type de chaque fonction.

Exemples

```
fun x -> x + 1
```

Inférence et vérification de types

- Il **infère** (calcule) le type de chaque valeur demandée.
- Il indique les **erreurs**.

Lire chaque diagnostic !

- Exemple : on ne peut pas concaténer par `^` deux entiers.
- OCaml **infère** aussi le type de chaque fonction.

Exemples

```
fun x -> x + 1
```

```
fun x -> x + 1.
```

Inférence et vérification de types

- Il **infère** (calcule) le type de chaque valeur demandée.
- Il indique les **erreurs**.

Lire chaque diagnostic !

- Exemple : on ne peut pas concaténer par `^` deux entiers.
- OCaml **infère** aussi le type de chaque fonction.

Exemples

```
fun x -> x + 1
```

```
fun x -> x + 1.
```

```
fun x -> x +. 1.
```

Polymorphisme

- OCaml infère **le type le plus général possible**.

  Exemples

  ```
  fun x -> x
  ```

  Quel est le type de cette fonction ?

Polymorphisme

- OCaml infère **le type le plus général possible**.

  Exemples

```
fun x -> x
```

  Quel est le type de cette fonction ?

```
fun x y -> (x, y)
```

  Et de celle-ci ?

Polymorphisme

- OCaml infère **le type le plus général possible**.

  Exemples

```
fun x -> x
```

  Quel est le type de cette fonction ?

```
fun x y -> (x, y)
```

  Et de celle-ci ?

```
fun x y -> x y
```

  Et enfin de celle-ci ?

Constructeurs de types

Il existe plusieurs façons de construire de nouveaux types :

- Chaque fonction a un type.
- Listes.
- Produit cartésien.
- Types union.

Plan

La liaison

On peut lier une valeur à un nom par la construction `let`.

```
let x = 7
let y = x * x
let y = 10 * y + 1
```

- Une fois une valeur liée à un nom, le nom peut être utilisé à la place de la valeur.
- Une nouvelle liaison avec le même nom masque la précédente.

La liaison

On peut lier une valeur à un nom par la construction `let`.

```
let x = 7
let y = x * x
let y = 10 * y + 1
```

- Une fois une valeur liée à un nom, le nom peut être utilisé à la place de la valeur.
- Une nouvelle liaison avec le même nom masque la précédente.

Un `let` n'est **pas une expression**.

La construction let ... in

- Il est parfois utile de faire des liaisons temporaires dans un calcul
- La construction `let ... in` le permet.
- Contrairement à `let`, cette construction **est une expression**.

La construction let ... in

- Il est parfois utile de faire des liaisons temporaires dans un calcul
- La construction `let ... in` le permet.
- Contrairement à `let`, cette construction **est une expression**.

```
(* une expression qui s'évalue en 49*)
let x = 7 in x * x
```

La construction let ... in

- Il est parfois utile de faire des liaisons temporaires dans un calcul
- La construction `let ... in` le permet.
- Contrairement à `let`, cette construction **est une expression**.

```
(* une expression qui s'évalue en 49*)
let x = 7 in x * x
```

```
(* une liaison de l'expression précédente: *)
(* y vaut 49 *)
let y =     let x = 7 in x * x
```

Plan

Récursivité : rappel et syntaxe

Sous Emacs...