La programmation fonctionnelle

...

Orleans Tech - Stéphane Legrand - 26 avril 2016

Le langage OCaml

ocaml.org try.ocamlpro.com

- INRIA (Institut National de Recherche en Informatique et en Automatique)
- Première version en 1996
- Communauté restreinte mais active
- Compilé : natif, bytecode et Javascript
- Fortement typé
- Fonctionnel mais aussi impératif et objet

Les bases

- Définition d'une variable

```
let toto = 42
```

- Définition d'une fonction

```
let add x y = x + y
```

- Appel d'une fonction

```
let seven = add 4 3
```

Immutabilité : variables non modifiables

```
let x = 10
let f y = x + y
let x = 20
f 100
```

renvoie 110, et non pas 120

- Ligne 3 : on a construit un nouveau x
- Ce second x “cache” le premier

Immutabilité : listes non modifiables

```
let l = [1; 2; 3]
```
liste d'entiers

aucun moyen de modifier directement un élément de `l`

```
let l = 4 :: l
```
[4; 1; 2; 3]

```
let l = l @ [4; 5]
```
[4; 1; 2; 3; 4; 5]

- A chaque fois, on construit une nouvelle liste
- Les listes `l` ne sont pas modifiées, elles sont juste “cachées”

Immutabilité : enregistrements non modifiables

```
type t = { name : string ; age : int }

let person = { name = “titi” ; age = 42 }

let older = { person with age = person.age * 2 }
```

- Nouvel enregistrement `older` à partir de `person`
- L’enregistrement `person` n’est pas modifié

Récursivité

```
let rec suml acc = function
      []          -> acc
    | e :: tail -> suml (acc + e) tail

suml 0 [1; 2; 3]
```

1. suml (0 + 1) [2; 3]
2. suml (1 + 2) [3]
3. suml (3 + 3) [] renvoie acc = 6

- Pas de boucle `for` ni de `while`

Composition de fonctions

les bases

- Fonction en paramètre

```
let f g x = (g x) + (g 10)
```

- Fonction qui retourne une fonction

```
let add10 x = x + 10

ou    let add10 = fun x -> x + 10

ou    let add x y = x + y
      let add10 = add 10   application partielle
```

Composition de fonctions

`filter et map`

```
let l = [1; 2; 3; 4]

let even v =
    v mod 2 = 0
```

vrai si v est pair, faux sinon

```
List.filter even l
```

renvoie liste des nb pairs = [2; 4]

```
List.map (fun e -> e + 10) l
```

renvoie [11; 12; 13; 14]

Composition de fonctions

`fold_left`

```
List.fold_left (fun acc e -> acc + e) 0 [1; 2; 3]
```

`(fun acc e -> acc + e)`	fonction appliquée
`0`	valeur initiale acc(umulateur)
`[1; 2; 3]`	liste utilisée

⇔ f (f (f 0 1) 2) 3 = ((0 + 1) + 2) + 3 = 6

Composition de fonctions

opérateur `|>`

```
let l = [1; 2; 3; 4]

let x = List.filter even l
List.map (fun e -> e * 2) x
```

liste nb pairs = [2; 4]
nombres * 2 = [4; 8]

⇔

```
List.filter even l
|> List.map (fun e -> e * 2)
```

liste nb pairs = [2; 4]
nombres * 2 = [4; 8]

```
x |> f      ⇔ f x
x |> f |> g ⇔ g (f x)
```

Typage

```
let x = 10 + "99"
```

Error: This expression has type string but an expression was expected of type int

```
let l = [1; "x"]
```

Error: This expression has type string but an expression was expected of type int

```
let f v =
  if v then
    (fun x y -> x + y)
  else (fun x -> x + 1)
```

Error: This expression has type int but an expression was expected of type int -> int

Typage

```
Printf.printf "v = %s" 10
```

Error: This expression has type int
but an expression was expected of
type string

```
let c = cm 1.0
let k = km 50.0
let total = c + k
```

Error: This expression has type
[`Km] Measure.t
but an expression was expected of
type [`Cm] Measure.t
…

Exemple

```
type data = {
    id : string ;                    identifiant client
    discount : float -> float        fct calcule un rabais
}

type customer =
    | Individual of data
    | Company of data
```

Exemple

```
let c1 = Individual {
    id = “toto”;
    discount = fun v -> v *. 0.05
}

let c2 = Company {
    id = “google”;
    discount = fun v -> v *. 0.15
}
```

Exemple

```
let l = [(c1, 60.25); (c2, 100.00)]
```
(client, somme)

```
let f (c, sum) = match c with
    | Individual d ->
          if d.id = “toto” then (c, sum *. 2.)
          else (c, sum -. d.discount sum)
    | Company d -> (c, sum -. d.discount sum)
```
$$$

```
List.map f l
```
[(c1, 120.5); (c2, 85)]

Plein d'autres choses encore...

- Modules
  - paramétrés par d'autres modules
  - manipulables comme des valeurs
    - en paramètres d'une fonction
    - renvoyés par une fonction
- Interfaces
- Types génériques
- Gestion des packages
- ...

Pour le web

- Projet Ocsigen : ocsigen.org
  applications web côté serveur et client
    démos : github.com/slegrand45/examples_ocsigen
- Webmachine : github.com/inhabitedtype/ocaml-webmachine
  API REST

- Elm : elm-lang.org
  langage fonctionnel dédié côté client

Style de programmation

- Peut essayer en Javascript par exemple, avec complément
  - Immutable.js : facebook.github.io/immutable-js/
- Mais demande de la discipline !
- Avec un langage dédié
  - plus facile
  - plus optimisé
  - difficile, voire impossible de “tricher”

En résumé

- Fonctions, fonctions, fonctions et... fonctions
- On gère des valeurs et non pas des états
- Programmation par types
- Pas plus difficile, juste différent

Merci

Questions ?