Programmation fonctionnelle

Compléments pour le projet

EISTI – CPI2

2018 – 2019

Signature d’un module

Signature d'un module : définition

Ensemble des valeurs accessibles ***de l'extérieur dudit module***

Syntaxe

```
module NomModule :
 sig    (* Declaration des valeurs publiques *)
   val valeur1 : typeValeur1
 end =
 struct (* Implementation des valeurs *)
   let valeur1 = ...
 end ;;
```

- Dans la section `sig ... end`, les déclarations ne se terminent pas par `;;`.
- Si la section `sig ... end` n'est pas présente, toute fonction définie dans la section `struct ... end` est accessible de l'extérieur du module.
- Il est possible d'avoir de définir dans la section `struct ... end` des valeurs autres que celles définies dans la section `sig ... end`.

Signature d'un module : exemple 1/2

Fichier `fact.ml`

```
module Factorielle =
  struct
    let rec fact n = if (n <= 0) then 1 else n * fact (n - 1) ;;
  end ;;
```

Utilisation

```
# #use "fact.ml" ;;
module Factorielle : sig val fact : int -> int end
# Factorielle.fact 4 ;;
- : int = 24
```

Important

Si la signature n'est pas explicitée, **CAML la construit à partir des éléments implémentés**.

Signature d'un module : exemple 2/2

Fichier `fact.ml`

```
module Factorielle :
  sig
    val fact : int -> int
  end =
  struct
    let rec factAcc n acc =
      if (n <= 0) then acc else factAcc (n - 1) (n * acc) ;;

    let fact n = factAcc n 1 ;;
  end ;;
```

Utilisation

```
# #use "fact.ml" ;;
module Factorielle : sig val fact : int -> int end
# Factorielle.fact 4 ;;
- : int = 24
# Factorielle.factAcc 4 1 ;;
Error: Unbound value Factorielle.factAcc
```

Le format `ocamldoc` pour les commentaires

Format `ocamldoc`

Présentation

- Utilisé pour l'outil de génération de documentation `ocamldoc`
- Format inspiré de *Javadoc* (intégré au système de développement Java depuis sa création)

Classification

- Outil intégré :
  - *Javadoc* pour Java
  - `ocamldoc`
  - *Scaladoc* pour Scala
- Outil externe à l'usage de languages ne bénéficiant pas de systèmes intégrés
  - *Doxygen* : très versatile (FORTRAN, C, C++, C#, PHP, Python, Java...)...
  - ...

EIST

Format `ocamldoc` : règles

Délimitation

- Commentaire normal : `(* ... *)`
- Commentaire `ocamldoc` : `(** ... *)`

Où placer le commentaire ?

Juste avant la valeur documentée :

- *ligne de déclaration* **`module`** (définition d'un module)
- *ligne de déclaration* **`val`** (dans la signature d'un module)
- *ligne de déclaration* **`let`** (dans un module. . . ou non)

Format ocamldoc : syntaxe

Documentation des valeurs non fonctionnelles

- Première ligne : rapide descriptif
- Lignes suivantes : descriptif détaillé

Exemple

```
(** Valeur de test
  * Ceci est une valeur de test.
  *)
val test = 0
```

Format `ocamldoc` : syntaxe

Documentation des valeurs fonctionnelles

- Première ligne : rapide descriptif
- Lignes suivantes : descriptif détaillé
- Lignes avec @ : éléments constitutifs de la fonction

Exemple

```
(** Factorielle
  * Cette fonction calcule la factorielle.
  * @param n entier
  * @return la factorielle de [n]
  *)
val fact : int -> int
```

Format `ocamldoc` : « @-tags »

Quelques « @-tags » utiles

- **`@author`** *`nom`*
  - *`nom`* : auteur de la fonction (ou du module)
- **`@param`** *`id desc`*
  - *`id`* : nom du paramètre formel
  - *`desc`* : description du paramètre formel
- **`@return`** *`desc`*
  - *`desc`* : description de ce qui est renvoyé par la fonction

Références dans la documentation officielle

- `ocamldoc` : section 15, pages 235–255
- liste des « @-tags » : section 15.2.5 page 252

EISTI

Le module **Random**

Module **Random** de génération pseudo-aléatoire

Quelques fonctions utiles

- **Random**.bool() : booléen aléatoire
- **Random**.int n : entier aléatoire entre 0 et n-1
  n doit être strictement positif
- **Random**.float x : réel aléatoire entre 0 et x
  x est de signe quelconque

Pour ces trois fonctions, toutes les valeurs générées sont **équiprobables**.

Références dans la documentation officielle

- section 21.28, pages 441-443

À paramètres identiques, résultats différents

⇒ Fonctions non pures

⇒ Hors du paradigme fonctionnel

EIST

EISTI

Retour sur le module `List`

Retour sur le module **`List`**

`List``.sort`

`List.sort`

Principe

- On se donne une fonction de comparaison sur les éléments de la liste.
- On trie les éléments de la liste selon cette fonction.

Signature de `List.sort`

- `cmp : 'a -> 'a -> int`
- `l : 'a list`

$\Rightarrow$ `List.sort cmp l : 'a list`

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

`List.sort`

Détails sur la fonction de comparaison `cmp`

`cmp x1 x2` renvoie un entier :

- nul si `x1` et `x2` sont « égaux » ;
- strictement positif si `x1` est « strictemement supérieur » à `x2` ;
- strictement négatif si `x1` est « strictemement inférieur » à `x2`.

Exemple

```
# let cmp x1 x2 = (fst x1) - (fst x2) ;;
val cmp : int * 'a -> int * 'b -> int = <fun>

# List.sort cmp [(2, "a") ; (1, "b") ; (3, "c")] ;;
- : (int * string) list = [(1, "b"); (2, "a"); (3, "c")]
```

Variantes de `List.sort`

- documentation officielle : section 21.18 "Sorting", page 416

Retour sur le module **`List`**

`List``.assoc`

`List.assoc`

Principe

- On se donne une clé.
- On cherche dans une liste de couples *(clé, valeur)* la première valeur associé à la clé.

Signature de `List.assoc`

- `key : 'a`
- `l : ('a * 'b) list`

$\Rightarrow$ `List.assoc key l : 'b`

```
List.assoc : 'a -> ('a * 'b) list -> 'b
```

`List.assoc`

`List.assoc` comme cas particulier de `List.find`

- `List.assoc key l`
  ≡ `snd (List.find (fun (k, v) -> k = key) l)`

⇒ Provoque l'exception `Not_found` si key n'existe pas

Exemple

```
# List.assoc 1 [(2, "a") ; (1, "b") ; (3, "c")] ;;
- : string = "b"

# List.assoc 0 [(2, "a") ; (1, "b") ; (3, "c")] ;;
Exception: Not_found.
```

Retour sur le module `List`

. . . et bien d'autres fonctions !

- des variantes à deux listes pour les fonctions vues en cours ;
- conversions liste de couples vers couple de listes ;
- . . .

Références dans la documentation officielle

- section 21.18, pages 412-416

Opérateurs d’application

Opérateur (|>)

Exemple

```
# let sqrt_int n = sqrt (float_of_int n) ;;
val sqrt_int : int -> float
# let sqrt_int n = n |> float_of_int |> sqrt ;;
val sqrt_int : int -> float
```

- `x |> f` ≡ `f x`
- `x |> f |> g` ≡ `g (f x)`

Signature

- `x : 'a`
- `f : 'a -> 'b`

⇒ `x |> f : 'b`

```
(|>) : 'a -> ('a -> 'b) -> 'b
```

Opérateur (@@)

Exemple

```
# let sqrt_int n = sqrt (float_of_int n) ;;
val sqrt_int : int -> float
# let sqrt_int n = sqrt @@ float_of_int @@ n ;;
val sqrt_int : int -> float
```

- `f @@ x` ≡ `f x`
- `g @@ f @@ x` ≡ `g (f x)`

Signature

- `f : 'a -> 'b`
- `x : 'a`

⇒ `f @@ x : 'b`

`(@@) : ('a -> 'b) -> 'a -> 'b`