OCaml en trois pages

François Pottier

Les numéros (Remarque, Détail, Exercice...) sont des références au poly INF441.

Fonctions et définitions de variables

Fonctions. (Détail 1.16.) Une fonction OCaml a toujours un argument et un résultat.

```
fun x -> x + 1                          (* fonction anonyme *)
let f = fun x -> x + 1                  (* fonction nommée *)
let f x = x + 1                         (* équivalent *)
let y = f 42                            (* appel de fonction *)
```

L'argument et/ou le résultat peut être un tuple :

```
let f (x, y) = (x + y, x - y)           (* définition *)
let (v, w) = f (4, 4)                   (* appel *)
```

L'argument et/ou le résultat peut être le tuple vide `()` (Remarque 1.3) :

```
let hello () = print_endline "Hello"
let be (polite : bool) = if polite then hello () else ()
```

Le résultat peut être une fonction (l'argument aussi, mais ce n'est pas illustré ici) :

```
let f = fun x -> fun y -> x + y         (* forme explicite *)
let f x = fun y -> x + y                (* équivalent *)
let f x y = x + y                       (* équivalent *)
let z = f 4 4                           (* appel *)
```

Une fonction peut être récursive :

```
let rec sum n = if n = 0 then 0 else sum (n-1) + n
```

On peut déclarer le type des arguments et du résultat :

```
let f (x : int) (y : int) : int = x + y
```

Définitions locales. (Remarque 1.6.) L'argument d'une fonction est une variable locale. Les variables introduites dans les branches d'un `match` (voir plus bas) sont également des variables locales. Enfin, la construction `let x = ... in ...` déclare et initialise une variable locale.

```
let x = 4 in x + x                      (* ceci est une expression *)
let x : int = 4 in x + x
```

Définitions globales. La phrase `let x = ...` définit la variable `x`, qui est visible dans toute la suite. Noter l'absence du mot-clef `in`.

```
let x = 3                               (* ceci est une phrase *)
let f = fun x -> x + 1                  (* ceci aussi *)
let f x = x + 1                         (* équivalent *)
```

Données

Types algébriques. (Poly §1.1.2. Remarque 1.4. Détails 1.13, 1.14.) On nomme les différentes étiquettes possibles (par exemple `Leaf`, `Node`) et pour chacune on indique les types des champs.

```
type tree = Leaf of char | Node of tree * tree
```

Allocation d'un bloc. (Poly §1.1.2.) On écrit l'étiquette, suivie de la valeur de chaque champ, avec parenthèses et virgules s'il y a plusieurs champs.

```
Leaf ('A')
Node (Leaf ('A'), Leaf ('B'))
```

Accès à un bloc. (Poly §1.1.2.) La construction **`match`** analyse l'étiquette. Dans chaque branche, on introduit pour chaque champ une variable locale, dont on choisit le nom.

```
match t with Leaf (c) -> ... | Node (t0, t1) -> ...
```

Types enregistrements. (Détail 1.11.) Ce sont des types algébriques à une seule branche. Dans ce cas, les champs sont nommés, et peuvent être modifiables.

```
type point = { mutable x: int; mutable y: int }
let p = { x = 3; y = 4 }                    (* allocation d'un bloc *)
let norm p = p.x * p.x + p.y * p.y          (* lecture de champs *)
let move p dx dy =                          (* mise à jour de champs *)
  p.x <- p.x + dx; p.y <- p.y + dy
```

Paires et tuples. (Détail 1.12.) Ce sont également des types algébriques à une seule branche. On peut les nommer en définissant une abréviation de types (Remarque 1.7).

```
type point = int * int                      (* abréviation de types *)
let p = (3, 4)                              (* allocation d'un bloc *)
let norm (x, y) = x * x + y * y             (* accès au bloc *)
```

Options et listes. (Remarque 1.4, Exercice 1.4.) Ce sont des types algébriques prédéfinis. Les listes bénéficient de notations spéciales : la liste vide s'écrit `[]` ; une liste contenant l'élément `x` suivi de la sous-liste `xs` s'écrit `x :: xs`. Voir le module `List` de la bibliothèque.

```
type 'a option = None | Some of 'a          (* prédéfini *)
type 'a list = [] | :: of 'a * 'a list      (* prédéfini *)
1 :: 2 :: 3 :: []                           (* une liste à 3 éléments *)
[1; 2; 3]                                   (* équivalent *)
```

Références. (Exercice 1.2.) C'est un type enregistrement prédéfini. Il représente un bloc de mémoire à un seul champ modifiable. Il bénéficie de notations spéciales.

```
type 'a ref = { mutable contents: 'a }      (* prédéfini *)
let r : int ref = ref 0                     (* allocation *)
!r                                          (* lecture *)
r := 42                                     (* écriture *)
```

Tableaux. (Détail 1.8) Le type `'a array` est prédéfini. Il représente un bloc de mémoire à n champs modifiables, tous de même type `'a`. Il bénéficie lui aussi de notations spéciales. Voir le module `Array` de la bibliothèque.

```
let a : char array = Array.make 100 'X'     (* allocation *)
a.(32)                                      (* lecture *)
a.(42) <- 'Y'                               (* écriture *)
Array.length a                              (* longueur *)
```

Organisation des fichiers

Chaque fichier `.ml` forme une « structure » et chaque fichier `.mli` forme une « signature ». Le mot « structure » est essentiellement synonyme de « module ».

Structures. (Poly §2.1.) Dans un fichier `.ml`, on place des déclarations de types et de valeurs.

```
type set = int list                (* un type *)
let empty = []                     (* une constante *)
let insert x xs = x :: xs          (* une fonction *)
```

Signatures. (Poly §2.1.) Dans un fichier `.mli`, on place des définitions et déclarations de types ainsi que des déclarations de valeurs. Elles décrivent (une partie de) ce qui est défini dans le fichier `.ml` correspondant.

```
type set                           (* un type abstrait *)
val empty: set                     (* une constante *)
val insert: int -> set -> set      (* une fonction *)
```

Structures et signatures explicites ; foncteurs

Structures nommées (Poly §3.3.1. Détail 3.4.) On peut (dans un fichier `.ml`) définir une structure et lui donner un nom.

```
module I = struct                  (* une structure nommée I *)
  type t = int                     (* un type *)
  let compare (x : t) (y : t) =    (* une fonction *)
    if x < y then -1 else if x > y then 1 else 0
end
```

Signatures nommées (Poly §3.3.1. Détail 3.5.) On peut (dans un fichier `.ml` ou `.mli`) définir une signature et lui donner un nom.

```
module type Ordered = sig          (* une sig. nommée Ordered *)
  type t                           (* un type *)
  val compare: t -> t -> int       (* une fonction *)
end
```

Définition d'un foncteur (Poly §3.1.) Comme une définition de fonction ordinaire, une définition de foncteur donne le nom et la signature de l'argument, et puis donne le résultat du foncteur, qui est une structure.

```
module Set (E : Ordered)           (* le foncteur Set attend un arg. E, *)
= struct                           (* et définit: *)
  type set = E.t list              (* un ensemble = une liste triée *)
  let empty = []                   (* l'ensemble vide *)
  let rec insert x xs =            (* l'insertion dans un ensemble *)
    match xs with
    | [] ->
        [x]
    | y :: ys ->
        if E.compare x y <= 0
        then x :: xs
        else y :: insert x ys
end
```

Utilisation d'un foncteur (Poly §3.1.) On appelle un foncteur un peu comme on appelle une fonction. Par exemple, pour obtenir un module `ISet` qui implémente des ensembles d'entiers :

```
module IntegerSet = Set(I)
```