

Ce que nous avons vu...

Programmation Fonctionnelle

Structures de données, filtrage par motifs

David Teller

29/01/2007

- ▶ lancement d'OCaml
- ▶ utilisation d'OCaml comme calculatrice
- ▶ fonctions
- ▶ valeurs
- ▶ définition de variables
- ▶ système de types

Au programme du jour

- ▶ Listes
- ▶ définition de structures de données
- ▶ filtrage par motifs
- ▶ projets.

Prélude

Premiers pas avec le filtrage par motifs

Listes

Mettons les listes au travail

Interlude

Sujets de projets

Rapports de projets

Structures de données

Alias

Produit

Types récursifs

Modules

Exercice 9

Énoncé Définissez le singleton, de la même manière que ce qui précède.

Exemple :

```
* let singleton_3 = singleton 3;;
- : int -> bool = <fun>
* singleton_3 0;;
- : bool = false
* singleton_3 3;;
- : bool = true
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Correction

Une réponse possible et tout à fait correcte :

```
* let singleton x =
  function y ->
    if x=y then
      true
    else
      false ;;
val singleton : 'a -> 'a -> bool = <fun>
```

Variante plus courte :

```
* let singleton x =
  function y -> x=y ;;
val singleton : 'a -> 'a -> bool = <fun>
```

Encore plus court :

```
* let singleton x y = x=y ;;
val singleton : 'a -> 'a -> bool = <fun>
```

Minimale :

```
* let singleton = (=) ;;
val singleton : 'a -> 'a -> bool = <fun>
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Exercice 10

Énoncé Définissez de même l'intersection.

```
* let intersection f g =
  function x -> (f x) && (g x) ;;
val intersection : ('a -> bool) -> ('a -> bool) -> 'a -> bool = <fun>
```

Ou, de manière plus concise,

```
* let intersection f g x = (f x) && (g x) ;;
val intersection : ('a -> bool) -> ('a -> bool) -> 'a -> bool = <fun>
```

⏪ ⏩ ⏴ ⏵ ↺ 🔍

Prélude

Premiers pas avec le filtrage par motifs

Listes

Mettons les listes au travail

Interlude

Sujets de projets

Rapports de projets

Structures de données

Alias

Produit

Types récursifs

Modules

Interlude

⏪ ⏩ ⏴ ⏵ ↺ 🔍

La factorielle, encore, toujours

Nous avons déjà (re)(re)(re)vu la factorielle en OCaml :

```
* let rec factorielle n =
  if n < 2 then
    1
  else
    n * (factorielle (n-1));;
val factorielle : int -> int = <fun>
```

Maintenant, ce n'est pas la définition mathématique de la factorielle.

$$\begin{cases} \text{factorielle}(0) & = 1 \\ \forall n \geq 1, \text{factorielle}(n) & = n \cdot \text{factorielle}(n-1) \end{cases}$$

C'est reparti pour une factorielle

$$\begin{cases} \text{factorielle}(0) & = 1 \\ \forall n \geq 1, \text{factorielle}(n) & = n \cdot \text{factorielle}(n-1) \end{cases}$$

se traduit

```
* let rec factorielle = function
  | 0 -> 1
  | n -> n * (factorielle (n-1));;
```

Cette fonction *filtre* avec les motifs 0 (la constante 0) et n (n'importe quel nombre n).

Le filtrage par motif

Le filtrage par motifs est un outil puissant de la programmation fonctionnelle, qui permet de définir une fonction par cas.

```
function
  | premier_motif -> premier_cas
  | deuxieme_motif -> deuxieme_motif
  | troisieme_motif -> troisieme_cas
  | ... -> ...
```

Note Les motifs peuvent contenir des conditions ou/et des noms de variables !

Exemple

Définissons le "ou exclusif" :

```
* let xor a b = match (a,b) with
  | (true, true) -> false
  | (true, false) -> true
  | (false, true) -> true
  | (false, false) -> false;;
val xor : bool -> bool -> bool = <fun>
```

ou encore

```
* let xor a b = match (a,b) with
  | (true, false) | (false, true) -> true
  | _ -> false;;
val xor : bool -> bool -> bool = <fun>
```

ou encore (la "bonne" manière)

```
* let xor a b = match (a,b) with
  | (true, false) | (false, true) -> true
  | (true, true) | (false, false) -> false;;
val xor : bool -> bool -> bool = <fun>
```

À retenir sur le filtrage par motifs

- ▶ Énormément de fonctions emploient le filtrage par motifs.
- ▶ Syntaxes : `match... with` – ou, directement `function`.
- ▶ Un motif commence toujours par la barre `|`.
- ▶ Cette même barre peut servir à composer deux motifs.
- ▶ Les motifs sont essayés dans l'ordre.
- ▶ Le motif spécial `_` se prononce "don't care". Il accepte tout. À employer avec précautions.
- ▶ Un motif peut contenir des variables – mais aucune variable ne peut apparaître plus d'une fois.

Filtrage non-exhaustif

Que se passe-t-il ?

```
* let xor_incomplet a b = match (a,b) with
| (true, false) | (false, true) -> true
| (true, true) -> false;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(false, false)
val xor_incomplet : bool -> bool -> bool = <fst>
```

OCaml vous prévient que vous n'avez pas géré tous les cas. C'est le message qu'a affiché OCaml à beaucoup d'entre vous lorsque vous avez voulu définir `let modulo p 3 = ...` – vous ne définissez *que* le cas 3.

```
* xor_incomplet false false;;
Exception: Match_failure (**, 79, -71).
```

To be continued

Nous y reviendrons bientôt, avec les listes.

Semi-formellement

Definition (Liste)

Une liste d'éléments de type α est soit la liste vide `[]` soit une tête de type α suivie d'une queue, qui est à son tour une liste d'éléments de type α .

Quelques listes

```
* []:;           (=La liste vide*)
- : 'a list = []
* [1,2,3,4,5]:;
- : int list = [1; 2; 3; 4; 5]
* [1]:;         (=Une liste d'entiers qui ne contient que le nombre 1*)
- : int list = [1]
* 1::[]:;      (=Une liste commençant par 1 et continuée par la liste vide.*)
- : int list = [1]
```

Remarquez le α de la liste vide : le type des listes est polymorphe.

Symboles à retenir

- ▶ [] – la liste vide (de n'importe quel type)
- ▶ :: – l'ajout d'une tête à une liste

```
* 1::[]:;      (=Une liste commençant par 1 et continuée par la liste vide.*)
- : int list = [1]
* [1]::[]:;
- : int list list = [[1]]
This expression has type int but is here used with type int list
```

Et les motifs, dans tout ça ?

Les listes – comme tous les types de données *sommes* – s'intègrent parfaitement au filtrage par motifs.

```
* let liste_vide = fonction
  | [] -> true
  | _::_ -> false;;
val liste_vide : 'a list -> bool = <fun>

* let rec longueur_liste = fonction
  | [] -> 0
  | _::t -> 1 + (longueur_liste t);;
val longueur_liste : 'a list -> int = <fun>
```

Notez la récursivité.

```
* let liste_vide = fonction
  | [] -> true
  | _::_ -> false;;
val liste_vide : 'a list -> bool = <fun>

* let rec longueur_liste = fonction
  | [] -> 0
  | _::t -> 1 + (longueur_liste t);;
val longueur_liste : 'a list -> int = <fun>
```

À vous !

Exercice Comment écrire une fonction qui teste si tous les éléments d'une liste sont pairs ?

Exemple :

```
* tout_le_monde_est_pair [2;4;6;8]:;
- : bool = false
* tout_le_monde_est_pair [2;4;6;8]:;
- : bool = true
```

Parité

Une solution possible

```
# let rec tout_le_monde_est_pair = function
| h::t -> h mod 2 = 0 && (tout_le_monde_est_pair t)
| [] -> true;;
val tout_le_monde_est_pair : int list -> bool = <fex>
```

Ou encore une autre.

```
# let rec tout_le_monde_est_pair = function
| [] -> true
| h::t when h mod 2 = 0 -> (tout_le_monde_est_pair t)
| _::_ -> false;;
val tout_le_monde_est_pair : int list -> bool = <fex>
```

Quelques fonctions existantes

Q Que font `List.hd` et `List.tl` ?

A `List.hd` prend le premier élément d'une liste. `List.tl` prend la suite de la liste.

Si la liste est vide, les deux lancent une exception.

En fait, ces deux fonctions sont déconseillées, justement, à cause de l'exception possible. Le filtrage par motifs est généralement un meilleur choix, puisqu'il permet de gérer les deux cas possibles d'un coup – et il permet à OCaml de garantir que vous avez géré les deux cas possibles.

Encore une pour la route

Exercice Comparer deux listes.

```
# let rec comparer l1 l2 = match (l1, l2) with
| (h1::t1, h2::t2) when h1=h2 -> comparer t1 t2
| ([] , []) -> true
| _ -> false;;
val comparer : 'a list -> 'a list -> bool = <fex>
```

Un petit bilan

Vous savez

- ▶ construire une liste
- ▶ consulter le contenu d'une liste
- ▶ analyser la structure d'une donnée.

Il vous manque maintenant la capacité de définir de nouvelles structures de données.

Prélude

Premiers pas avec le filtrage par motifs

- Listes
- Mettons les listes au travail

Interlude

- Sujets de projets
- Rapports de projets

Structures de données

- Alias
- Produit
- Types récursifs
- Modules

Interlude

Interlude

Parlons de nouveau de vos projets.

Vous avez une semaine pour me proposer une idée de projet, décrite en un paragraphe.

Vous êtes encouragés à demander à un enseignant de vous encadrer pour toute la partie qui ne relève pas uniquement de la programmation.

Objectifs du projet

- ▶ Vous faire travailler par vous-même sur des problèmes "réels", tels que ceux que vous devrez traiter dans votre vie professionnelle.
- ▶ Vous faire mettre les mains dans le cambouis, faire toutes les erreurs possibles, de manière à ne pas les refaire plus tard (notamment pendant l'examen).
- ▶ Vous faire travailler la rédaction de documents scientifiques ou/et techniques.
- ▶ Vous montrer que vous en êtes capables.

En particulier, cela signifie que vous allez devoir passer de nombreuses heures à travailler sur le projet, hors des cours et des TDs.

Vous pourrez me contacter

Par mail David.Teller@univ-orleans.fr – inscrivez [Projet] dans le sujet de votre mail.

Dans mon bureau Le lundi après vos cours.

Déroulement du projet

1. Former un binôme.
2. Choisir un sujet.
3. Préciser le sujet et les objectifs.
4. Procéder à des recherches bibliographiques.
5. Découper le sujet en problèmes, sous-problèmes, etc.
6. Répartir les sous-problèmes dans le binôme.
7. Répartir les sous-problèmes en modules.
8. Définir les interfaces des modules.
9. Concevoir les protocoles de tests.
10. Concevoir les algorithmes.
11. Implanter les algorithmes.
12. Tester.
13. Corriger les bugs.
14. Documenter.
15. Ajouter des fonctionnalités et retourner en 3.
16. Rédiger le rapport.

Sujets mathématiques

- Traceur de courbes** Écrire un programme qui comprend les définitions mathématiques de courbes définies implicitement (par $y = f(x)$) ou explicitement (par $f(x, y) = 0$) et fait afficher à l'écran la courbe ou la sauvegarde comme une liste de coordonnées de points ou encore comme une image.
- Approximateur de courbes** Écrire un programme qui, à partir d'une courbe, essaye de construire une équation qui donnerait une courbe proche.
- Afficheur de fractales** Écrire un programme qui dessine les ensembles de Julia, de Mandelbrot et autres.



Sujets de mathématiques appliquées

- Cryptographie à clef publique** Écrire un programme capable de construire un couple clef publique/clef privée, de chiffrer un message à partir d'une clef publique et de le déchiffrer à partir de la clef privée correspondante.
- Partage de secret** Écrire un programme capable de diviser un message en n secrets, de manière à ce que l'accord de m personnes soit nécessaire et suffisant pour retrouver le message.



Sujets d'Intelligence Artificielle

- Perceptron** Écrire un programme de reconnaissance de formes, de lettres... capable d'apprendre !
- Langue de bois** Écrire un générateur automatique de retranscriptions de débats télévisés.
- Contrôle de robots** Écrire un programme capable de trouver un moyen de faire bouger un robot (ou un bras robotique, ou une pince...) d'un point A à un point B , en gérant les obstacles.



Jeux

- Démineur** Vous connaissez le concept.
- Tetris** Vous connaissez le concept.
- Puissance 4** Un jeu de Puissance 4 avec une interface graphique et soit une Intelligence Artificielle, soit la possibilité de jouer à deux à travers le réseau.
- Touché coulé** À deux à travers le réseau.
- Jeu d'aventures** Un jeu d'aventures à la Myst.
- Affichage d'échecs** À partir d'une retranscription d'une partie d'échecs, afficher la partie, en 3 dimensions.



Utilitaires

Explorateur de fichiers Un joli explorateur de fichiers, capable d'afficher l'arborescence des répertoires, etc.

Gestionnaire de mots de passe Un programme qui se souvient de vos mots de passe – mais les conserve cryptés sur le disque, et protégés eux-mêmes par un mot de passe.

D'autres idées ?

Je rappelle que les projets sont à faire en binôme.
Vous avez une semaine pour former les binômes et choisir un sujet.
Si vous avez d'autres idées de sujets, n'hésitez pas à m'en parler.

Oui, mais et le rapport ?

Vous avez un rapport à écrire.

Objectifs du rapport

Le rapport sert à
permettre à l'enseignant de déterminer si vous avez bien fait votre travail

vous enseigner comment rédiger un rapport technique – il est donc prévu pour être lu par quelqu'un qui n'est pas nécessairement familier avec le domaine et qui ne connaît initialement rien du projet.

Structure

- Résumé** Résumer en une demi-page de quoi parle le document.
- Remerciements** Si nécessaire.
- Table des matières**
- Introduction** Intéresser le lecteur, présenter rapidement le contexte, le problème, ce qui devait être fait, ce qui a été fait.
- Description du travail** Spécifications, choix scientifiques et techniques, découpage en sous-problèmes, répartition des tâches, difficultés, solutions, algorithmes...
- Conclusion** Réponses aux questions de l'introduction, évaluation des choix, points forts et points faibles du projet, suites possibles.
- Bibliographie** Tous les ouvrages consultés ou/et cités.
- Annexes** Manuel du programme, documentation technique du programme, listing du programme.

L'introduction et la conclusion

Un lecteur pressé ne lira *que* l'introduction et la conclusion.
Un examinateur relira *uniquement* l'introduction et la conclusion avant de mettre sa note finale.

- ▶ évitez les banalités ("l'informatique, c'est super-important")
- ▶ il y a toujours quelque chose de pertinent à dire
- ▶ intéressez le lecteur
- ▶ considérez que le lecteur est intelligent mais n'est pas un spécialiste

Comptez entre 1/2 et 3 pages d'introduction, entre 1/2 page et 1 page de conclusion.

Le corps (1)

C'est la partie la plus longue du texte.
Cette partie doit permettre à l'examineur

- ▶ de déterminer si vous avez travaillé sérieusement, aussi bien sur la partie programmation que sur les autres aspects (recherches bibliographiques, etc.)
- ▶ de déterminer les difficultés que vous avez rencontrées
- ▶ de déterminer l'intelligence de vos choix et de vos initiatives – un projet sans choix ni initiatives sera forcément moins bien vu

Le corps (2)

Cette partie est aussi là pour vous éventuels successeurs :

- ▶ pour qu'ils comprennent le début de l'histoire
- ▶ pour qu'ils comprennent l'intérêt de vos choix
- ▶ pour qu'ils ne s'égarent pas sur de fausses pistes, pour qu'ils ne refassent pas vos erreurs.

Suggestion de plan du corps

Il n'est plus nécessaire de présenter le problème à résoudre, ni l'intérêt de ce problème. Vous avez déjà fait cela dans le cadre de l'introduction.

1. Contexte
2. Spécifications
3. Conception
4. Protocole de tests
5. Implantation
6. Résultat des tests
7. Retour en 2.



Contexte

Mettre le projet dans le contexte.

- ▶ Qu'est-ce qui existe déjà dans le domaine ?
- ▶ Quels sont les intérêts et les inconvénients de chaque approche ?
- ▶ Quelle est l'originalité de votre approche ?
- ▶ Si vous vous inspirez des travaux de quelqu'un d'autre, citez-les.



Spécifications

La liste des choses que votre projet doit faire.

- ▶ S'il y a un travail scientifique ou une théorie sous-jacente, détaillez-la.
- ▶ Détaillez les règles du jeu, ce que votre programme doit faire, ce qu'il ne doit pas faire...
- ▶ Si vous utilisez des bibliothèques de programmation, précisez lesquelles, ainsi que vos critères de choix.

À ce stade-là, on ne parle pas encore de fonctions (sauf mathématiques), de programme, de types...



Conception

Transformation du problème en une liste de tâches de programmation.

- ▶ Découpage de votre logiciel en fonctionnalités.
- ▶ Structures de données.
- ▶ Découpage des fonctionnalités en fonctions.
- ▶ Répartition des tâches entre étudiants.
- ▶ Présentation brève des algorithmes principaux.



Protocole de tests

Une fois que votre logiciel sera terminé, comment vérifierez-vous qu'il fonctionne correctement ?

- ▶ Liste des tests de fonctionnement, avec les résultats que vous attendez.
- ▶ Liste des tests de performances, avec les résultats que vous attendez.
- ▶ Si vous êtes conduits à écrire d'autres programmes pour lancer ou/et exploiter les tests, parlez-en ici.

Le protocole de tests est fondamental !

Implantation

- ▶ Les algorithmes, maintenant présentés de manière détaillée, avec autant d'illustrations et d'exemples que nécessaire pour que n'importe qui les comprenne. Pour chacun de vos algorithmes, s'il y a des effets de bord, précisez
 - ▶ dans quel état doit être le système pour s'assurer que votre algorithme fonctionne
 - ▶ dans quel état il laisse le système
- ▶ Les difficultés que vous avez rencontrées et la méthode que vous avez employé pour les résoudre (ou pas).
- ▶ Toute autre information pertinente.

Dans cette partie, vous parlez effectivement du code !

Résultat des tests

- ▶ Résultat détaillé des tests de fonctionnement – et vos conclusions.
- ▶ Résultat détaillé des tests de performances – et vos conclusions.
- ▶ Les modifications auxquelles ont conduit ces tests.

Notation

Préparation avant codage Travail scientifique, spécification, liste d'objectifs... – Environ 15% de la note.

Compétence Vos capacités à programmer, à faire face aux difficultés imprévues... – Environ 25% de la note.

Le logiciel Environ 30% de la note.

Rapport Environ 30% de la note.

Insistons sur un point

Utiliser des travaux existants, c'est bien.
Plagier des travaux existants, c'est 0.

Prélude

Premiers pas avec le filtrage par motifs

Listes

Mettons les listes au travail

Interlude

Sujets de projets

Rapports de projets

Structures de données

Alias

Produit

Types récursifs

Modules

Structures de données

Nous avons déjà vu

- ▶ les variables
- ▶ les n-uplets
- ▶ les listes.

Bien entendu, OCaml permet de définir des structures supplémentaires :

- alias** donner un nouveau nom à un type existant
- sommes** mettre une chose ou une autre dans une structure
- produits** mettre plusieurs choses à la fois dans une structure
- objets** l'équivalent des classes de Java, en plus puissant
- modules** l'équivalent des modules de Java, en plus puissant.

Alias

Il est possible de donner un nouveau nom à un type existant.

```
* type entier = int ::  
type entier = int  
* let f x = x + 1 ::  
val f : int -> int = <fun>  
* let f (x:entier) = x ::  
val f : entier -> entier = <fun>
```

Ce n'est pas très utile, mais retenez la syntaxe `type t = ..`.

Types sommes

On utilise les types *sommes* pour représenter l'union disjointes de types.

```
# type couleur = Treffe | Pique | Coeur | Carreau ::
type couleur = Treffe | Pique | Coeur | Carreau
# type valeur = Nombre of int | Valet | Dame | Roi ::
type valeur = Nombre of int | Valet | Dame | Roi
# type carte = Joker | Carte of couleur * valeur ::
type carte = Joker | Carte of couleur * valeur
```

Notez la syntaxe `type t = Constructeur | Constructeur | ...`
Constructeur of `t'` | Constructeur of `t''` |

Notez les majuscules à la définition de chaque *constructeur* – c'est presque la seule utilisation des majuscules en OCaml.

Filtres !

Les types sommes s'intègrent fort bien au filtrage par motifs.

```
# match carte with
| Joker -> "Le Joker, Batman n'est pas loin"
| Carte (Pique, Nombre 1) -> "As de pique, la carte de la mort"
| Carte (_, Nombre 1) -> "Un quelconque"
| _ -> "Une carte sans importance";;
```

À ce sujet

OCaml définit le très utile type `option` :

```
# type 'a option = None | Some of 'a ::
```

Ce type servir lorsqu'un algorithme peut échouer et n'avoir aucune réponse à donner.

Utilisons les options

Exemple

```
# let rec chercher_association a l = match l with
| [] -> None
| (f,v)::_ when f== a -> Some v
| _::l -> chercher_association a l
val chercher_association : 'a -> ('a * 'b) list -> 'b option = <fex>

# chercher_association 2 [(1,'un'); (2,'deux'); (3,'trois'); (4,'quatre'); (5,'cinq')
- : string option = Some "deux"

# chercher_association 7 [(1,'un'); (2,'deux'); (3,'trois'); (4,'quatre'); (5,'cinq')
- : string option = None
```

Produit

Le *produit* sert à combiner simultanément plusieurs informations en une seule.

OCaml accepte deux styles de produits :

produit cartésien vous avez déjà vu le produit cartésien (couples, triplets...)

enregistrements qui ressemblent plus aux classes Java, sans les méthodes ni le constructeur.

```
# type coordonnees = { x : int; y : int; z : int };
type coordonnees = { x : int; y : int; z : int };
# {x = 9; z = 10; y = 20};
- : coordonnees = {x = 9; y = 20; z = 10}
```

Notez que l'ordre n'intervient pas.



Distinction ?

Q Pourquoi OCaml offre-t-il deux types de produits ?

A Les couples, triplets... sont très utiles pour deux ou trois informations. Dès qu'il y a plus d'informations, on a vite tendance à ne plus savoir dans quel ordre elles doivent être mises – surtout si elles sont du même type.

Par opposition, les *enregistrements* sont plus ennuyeux à définir, mais ne dépendent pas de l'ordre dans lequel on note leur contenu.

Moralité Comme souvent avec OCaml, vous avez le choix entre plusieurs méthodes. Choisissez la plus adaptée.



Usage du temps

OCaml définit le temps à l'aide de la structure de donnée `tm`, comme suit :

```
type tm = {
  tm_sec : int;      (= Seconde 0..59 00   =)
  tm_min : int;     (= Minutes 0..59  =)
  tm_hour : int;    (= Heurs 0..23   =)
  tm_mday : int;    (= Day of month 1..31  =)
  tm_mon : int;     (= Month of year 0..12 =)
  tm_year : int;    (= Year from 1900  =)
  tm_wday : int;    (= Day of week (Sunday is 0) =)
  tm_yday : int;    (= Day of year 0..365  =)
  tm_isdst : bool;  (= Daylight time savings in effect =)
}

# qmtime (time ());
- : Unix.tm =
{tm_sec = 12; tm_min = 48; tm_hour = 21; tm_mday = 23; tm_mon = 0;
 tm_year = 107; tm_wday = 2; tm_yday = 22; tm_isdst = false}
```

Au fait...

Q Quel genre de type sont les listes ?

A Les listes sont un type somme – avec juste une syntaxe simplifiée parce qu'on les utilise en permanence.

```
# type 'a ma_liste = Vide | Contenu of 'a * 'a ma_liste;;
type 'a ma_liste = Vide | Contenu of 'a * 'a ma_liste;;
# Contenu (1, Contenu (2, Contenu (3, Vide)));
- : int ma_liste = Contenu (1, Contenu (2, Contenu (3, Vide)))
```

Une petite note au passage : le langage OCaml est modifiable. En particulier, on peut modifier la syntaxe pour simplifier l'usage de certaines fonctions ou ajouter des fonctions. Pour ce faire, l'outil s'appelle `Camlp4`.



Un mot sur les modules

Comme tous les langages modernes, OCaml supporte le découpage en modules.

Un module est un ensemble de

- ▶ types
- ▶ exceptions
- ▶ valeurs (y compris des fonctions)

qu'on a regroupés sous un nom commun.



Pour charger un module...

Un module est contenu dans un fichier `.cmo` ou `.cma`. Mettons `monmodule.cmo`.

Avant d'être utilisé, un module doit être chargé.

- ▶ si c'est un module standard, il est automatiquement chargé
- ▶ si vous utilisez le toplevel, lancez `ocaml monmodule.cmo` au lieu de `ocaml`
- ▶ si vous utilisez le compilateur, lancez `ocamlc monmodule.cmo` au lieu de `ocamlc`.



Pour utiliser un module...

On peut ouvrir le module, à l'aide de `open` :

```
* open List ;;
(* Pas de réponse *)
* for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
```

L'autre possibilité, conseillée, est de préfixer par le nom du module le nom du type/exception/valeur que vous voulez utiliser :

```
* List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
```



Pour faire un module...

Méthode simple :

1. rassembler des types, exceptions et valeurs dans un fichier (mettons `monmodule.ml`)
2. compiler ce fichier (avec `ocamlc -c monmodule.ml`)
3. laisser sécher.

Votre module est presque prêt à l'emploi. Il s'appelle `Monmodule`.



Prélude

Premiers pas avec le filtrage par motifs

- Listes
- Mettons les listes au travail

Interlude

- Sujets de projets
- Rapports de projets

Structures de données

- Alias
- Produit
- Types récursifs
- Modules

Interlude

OCaml, c'est bien

D'ailleurs, c'est si bien que nous avons vu presque toutes les bases du langage.

Après, il restera à mettre tout ça en pratique.

Du coup, voici déjà quelques conseils.

Conventions

OCaml fait la différence entre majuscules et minuscules :

minuscules noms de valeurs (fonctions y compris) et de types

majuscules noms de *constructeurs* et de modules

Insistons sur ce fait : il y a une différence *sémantique* entre majuscules et minuscules.

```
* let Majuscule x = x ;;      (= Il aurait fallu faire let minuscule x = x *)
Unbound constructor Majuscule
* type Majuscule = int ;;    (= Il aurait fallu faire type minuscule = int *)
Syntax error
* type t = minuscule of int ;; (= Il aurait fallu faire type t = Majuscule of int *)
Syntax error
```

Conventions supplémentaires

- ▶ si un nom de valeur ou de type contient plusieurs mots, séparez-les par des `_`
- ▶ par contre, n'utilisez pas de `_` dans les noms de constructeurs ou de modules
- ▶ n'utilisez pas d'accents dans les noms
- ▶ si vous utilisez un opérateur, mettez des espaces autour
- ▶ Pseudo-Loi de Landin : mettez en page vos programmes comme la signification du programme dépendait de cette mise en page.

Conseils

- ▶ Passer deux fois plus de temps sur un programme pour le rendre deux fois plus simple est un bon investissement
- ▶ divisez les grosses fonctions en petites fonctions
- ▶ si vous venez de faire un copier-coller dans le code, vous auriez probablement du écrire une fonction
- ▶ usez et abusez des *assertions* – elles sont souvent plus claires que les commentaires et elles aident à tester
- ▶ si le compilateur vous dit de faire attention à quelque chose (*warning*), faites-y attention !

Comment commenter

- ▶ Quand il y a une difficulté, commentez
- ▶ quand il n'y a pas de difficultés, ne commentez pas
- ▶ pas trop de commentaires à l'intérieur d'une fonction
- ▶ un commentaire qui n'apporte aucune information dérange
- ▶ si une fonction a un effet de bord, précisez dans quel état elle laisse le système
- ▶ si une fonction dépend d'un effet de bord, précisez dans quel état le système doit être au moment où on l'invoque.

À propos des noms

- ▶ le nom d'une valeur locale (`let x = ... in ...`) peut être court et abscons, personne ne s'en servira à part vous
- ▶ le nom d'une valeur globale (`let ma_fonction_qui_sert_a_ceci = ...`) peut être long et détaillé, ça ne gênera personne d'avoir une idée de à quoi sert la fonction – et puis si ça les gêne, ils écriront juste `let a = ma_fonction_qui_sert_a_ceci`

Autre conseils

- ▶ Les parenthèses servent à gérer les priorités – en cas d'ambiguïté, ajoutez des parenthèses
- ▶ le filtrage par motifs, c'est bon, mangez-en
- ▶ les exceptions, c'est bon aussi
- ▶ les structures de données, c'est encore mieux – presque tout algorithme peut bénéficier de bonnes structures de données
- ▶ ne laissez pas un filtrage par motifs incomplet.

Traits impératifs

- ▶ `for` existe et peut servir, si votre boucle n'est pas trop compliquée
- ▶ `while` existe, mais ne vous en servez pas
- ▶ n'utilisez les structures mutables qu'en dernier recours
- ▶ séparez clairement les fonctions qui ont des effets de bord de celles qui n'en ont pas

C'est fini pour les conseils

Fini pour le moment.

Repensez-y durant le semestre.

Au programme du prochain cours :

- ▶ des exceptions
- ▶ des exemples
- ▶ ... et des modules.