OCaml

Prise de contact et traits fonctionnels

Crédits

- Ce cours de Olivier Pons

Crédits

- Ce cours de Olivier Pons
- Cette page de la documentation officielle et les remarques à propos des parenthèses.

Crédits

- Ce cours de Olivier Pons
- Cette page de la documentation officielle et les remarques à propos des parenthèses.
- Cette introduction de Jean-Christophe Filliatre

Crédits

- Ce cours de Olivier Pons
- Cette page de la documentation officielle et les remarques à propos des parenthèses.
- Cette introduction de Jean-Christophe Filliatre
- Quelques mots sur la compilation

2 compilateurs (MP2I)

Il y a deux compilateurs en OCaml :

- `ocamlc` qui produit un *bytecode* ; un code compréhensible par un *interpréteur* appelé `ocamlrun`.
  Avantages : simplicité, portabilité et rapidité de compilation.

2 compilateurs (MP2I)

Il y a deux compilateurs en OCaml :

- `ocamlc` qui produit un *bytecode* ; un code compréhensible par un *interpréteur* appelé `ocamlrun`.
  Avantages : simplicité, portabilité et rapidité de compilation.
- `ocamlopt` qui compile directement en langage machine (l'exécution sera alors plus rapide qu'avec l'intrepréteur). Il accepte des fichiers de type `.ml` (fichiers de code) et `.mli` (fichiers d'interfaces).
  Il produit

2 compilateurs (MP2I)

Il y a deux compilateurs en OCaml :

- `ocamlc` qui produit un *bytecode* ; un code compréhensible par un *interpréteur* appelé `ocamlrun`.
  Avantages : simplicité, portabilité et rapidité de compilation.
- `ocamlopt` qui compile directement en langage machine (l'exécution sera alors plus rapide qu'avec l'intrepréteur). Il accepte des fichiers de type `.ml` (fichiers de code) et `.mli` (fichiers d'interfaces).
  Il produit
  - Un fichier objet `.o` en langage machine ;

2 compilateurs (MP2I)

Il y a deux compilateurs en OCaml :

- `ocamlc` qui produit un *bytecode* ; un code compréhensible par un *interpréteur* appelé `ocamlrun`.
  Avantages : simplicité, portabilité et rapidité de compilation.
- `ocamlopt` qui compile directement en langage machine (l'exécution sera alors plus rapide qu'avec l'intrepréteur). Il accepte des fichiers de type `.ml` (fichiers de code) et `.mli` (fichiers d'interfaces).
  Il produit
  - Un fichier objet `.o` en langage machine ;
  - Un fichier `.cmx` contenant des informations pour l'édition de lien et l'optimisation des relations entre les divers modules.

2 compilateurs (MP2I)

Il y a deux compilateurs en OCaml :

- `ocamlc` qui produit un *bytecode* ; un code compréhensible par un *interpréteur* appelé `ocamlrun`.
  Avantages : simplicité, portabilité et rapidité de compilation.
- `ocamlopt` qui compile directement en langage machine (l'exécution sera alors plus rapide qu'avec l'intrepréteur). Il accepte des fichiers de type `.ml` (fichiers de code) et `.mli` (fichiers d'interfaces).
  Il produit
  - Un fichier objet `.o` en langage machine ;
  - Un fichier `.cmx` contenant des informations pour l'édition de lien et l'optimisation des relations entre les divers modules.
  - Un fichier d'interface `.cmi` qui contient les signatures des types exportés vers d'autres unités de compilation.

Hello world (MP2I)

- Dans un fichier `hello.ml` écrivons

```
1 print_string "hello world\n"
2
```

Pas besoin de fonction `main()`. Pas besoin de parenthèse !

Hello world (MP2I)

- Dans un fichier `hello.ml` écrivons

```
1 print_string "hello world\n"
2
```

Pas besoin de fonction `main()`. Pas besoin de parenthèse !

- Compilons et exécutons

```
ocamlopt asup.ml -o hello
```

Hello world (MP2I)

- Dans un fichier `hello.ml` écrivons

```
1 print_string "hello world\n"
2
```

Pas besoin de fonction `main()`. Pas besoin de parenthèse !

- Compilons et exécutons

```
ocamlopt asup.ml -o hello
```

- Sans surprise, la commande `./toto` affiche

```
hello world
```

Hello world (MP2I)

- Dans un fichier `hello.ml` écrivons

```
1 print_string "hello world\n"
2
```

Pas besoin de fonction `main()`. Pas besoin de parenthèse !

- Compilons et exécutons

```
ocamlopt asup.ml -o hello
```

- Sans surprise, la commande `./toto` affiche

```
hello world
```

- Listons les fichiers de préfixe `hello`

```
$ ls hello*
hello  hello.cmi  hello.cmx  hello.ml  hello.o
```

Commentaires

Les commentaires en OCaml commencent par `(*` et finissent par `*)`.

```
(*on déclare x et on teste si il est pair ou impair*)
let x = 3 in x mod 2;;
(*On peut
faire des commentaires
sur plusieurs lignes*)
```

Application d'une fonction à ses arguments

- En OCaml, l'application d'une fonction s'écrit par **simple juxtaposition de la fonction et de son argument**. À la différence de la plupart des langages où l'application de `f` à `x` doit s'écrire `f(x)`, on se contente ici d'écrire `f x`.

Application d'une fonction à ses arguments

- En OCaml, l'application d'une fonction s'écrit par **simple juxtaposition de la fonction et de son argument**. À la différence de la plupart des langages où l'application de `f` à `x` doit s'écrire `f(x)`, on se contente ici d'écrire `f x`.
- Rien n'interdit d'écrire

```
1 print_string("hello world!\n")
```

mais les parenthèses autour de la chaîne de caractères sont tout simplement inutiles. Et c'est considéré comme inélégant.

Application d'une fonction à ses arguments

- En OCaml, l'application d'une fonction s'écrit par **simple juxtaposition de la fonction et de son argument**. À la différence de la plupart des langages où l'application de `f` à `x` doit s'écrire `f(x)`, on se contente ici d'écrire `f x`.
- Rien n'interdit d'écrire

```
1 print_string("hello world!\n")
```

  mais les parenthèses autour de la chaîne de caractères sont tout simplement inutiles. Et c'est considéré comme inélégant.
- En dehors des appels de fonctions, les parenthèses peuvent et doivent être utilisées lorsque les priorités des opérateurs l'exigent, comme dans l'expression `2*(1+3)`.

Affichage avec `printf`

- L'affichage en OCaml est un peu contraignant :

Affichage avec `printf`

- L'affichage en OCaml est un peu contraignant :
  - `print_string` pour les chaînes de caractères,

Affichage avec `printf`

- L'affichage en OCaml est un peu contraignant :
  - `print_string` pour les chaînes de caractères,
  - `print_int` pour les entiers,

Affichage avec `printf`

- L'affichage en OCaml est un peu contraignant :
  - `print_string` pour les chaînes de caractères,
  - `print_int` pour les entiers,
  - `print_float` pour les flottants...

Affichage avec `printf`

- L'affichage en OCaml est un peu contraignant :
  - `print_string` pour les chaînes de caractères,
  - `print_int` pour les entiers,
  - `print_float` pour les flottants...
- Heureusement, OCaml emprunte au C quelques habitudes comme la fonction `printf` du module `Printf`.

```
# Printf.printf "une chaîne : %s, un entier : %d, un
    flottant %f\n" "toto" 45 32.3;;
une chaîne : toto, un entier : 45, un flottant 32.300000
- : unit = ()
```

Présentation

- Le plus souvent, nous ne compilerons pas les codes `.ml`.

Présentation

- Le plus souvent, nous ne compilerons pas les codes `.ml`.
- Nous nous contenterons d'utiliser une *boucle interractive*. Il en existe plusieurs. Faisons un rapide tour d'horizon.

Dans un terminal

- OCaml est un langage compilé. Cependant il possède une *boucle interactive* (qui utilise un *interpréteur*) bien utile pour les tests.

Dans un terminal

- OCaml est un langage compilé. Cependant il possède une *boucle interactive* (qui utilise un *interpréteur*) bien utile pour les tests.
- Il suffit d'entrer `ocaml` dans un terminal.

```
ivan@fixe ~/.../OCAML$ ocaml
        OCaml version 4.05.0
```

Dans un terminal

- OCaml est un langage compilé. Cependant il possède une *boucle interactive* (qui utilise un *interpréteur*) bien utile pour les tests.
- Il suffit d'entrer `ocaml` dans un terminal.

```
ivan@fixe ~/.../OCAML$ ocaml
        OCaml version 4.05.0
```

- Des indications s'affichent et une invite de commande, le caractère `#`, invite l'utilisateur à entrer une phrase écrite dans la syntaxe OCaml

Dans un terminal

- Entrons une commande d'affichage :

```
# print_string "hello\n";;(*un commentaire*)
hello
- : unit = ()
# 1+2;;(*ne pas oublier les deux points-virgules*)
- : int = 3
```

Dans un terminal

- Entrons une commande d'affichage :

```
# print_string "hello\n";;(*un commentaire*)
hello
- : unit = ()
# 1+2;;(*ne pas oublier les deux points-virgules*)
- : int = 3
```

- Observons que le *type* des expressions est calculé (on dit *inféré*) et affiché. Ici, le type `unit` correspond au type `void` en C.

Dans un terminal

- Entrons une commande d'affichage :

```
# print_string "hello\n";;(*un commentaire*)
hello
- : unit = ()
# 1+2;;(*ne pas oublier les deux points-virgules*)
- : int = 3
```

- Observons que le *type* des expressions est calculé (on dit *inféré*) et affiché. Ici, le type `unit` correspond au type `void` en C.
- On quitte la boucle interactive avec la commande `exit(0);;`

Dans un terminal

- Entrons une commande d'affichage :

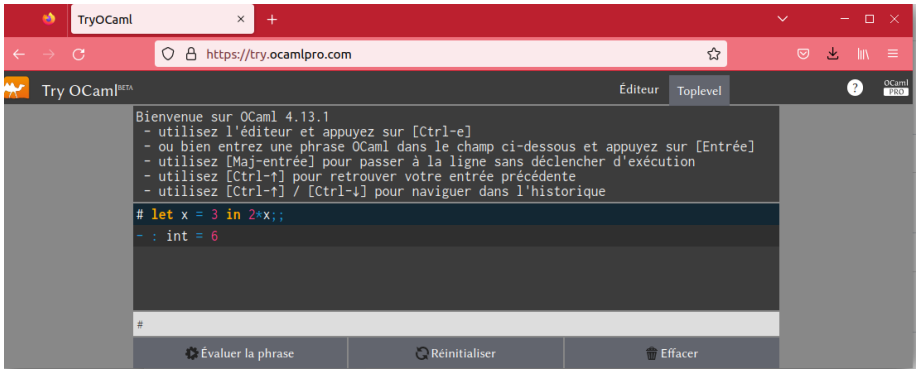
```
# print_string "hello\n";;(*un commentaire*)
hello
- : unit = ()
# 1+2;;(*ne pas oublier les deux points-virgules*)
- : int = 3
```

- Observons que le *type* des expressions est calculé (on dit *inféré*) et affiché. Ici, le type `unit` correspond au type `void` en C.
- On quitte la boucle interactive avec la commande `exit(0);;`
- Le double point-virgule sépare les différentes décalarations et expressions à évaluer.

Interpréteurs en ligne

Si on n'a pas encore installé OCaml sur sa machine, on peut utiliser un interpréteur en ligne :

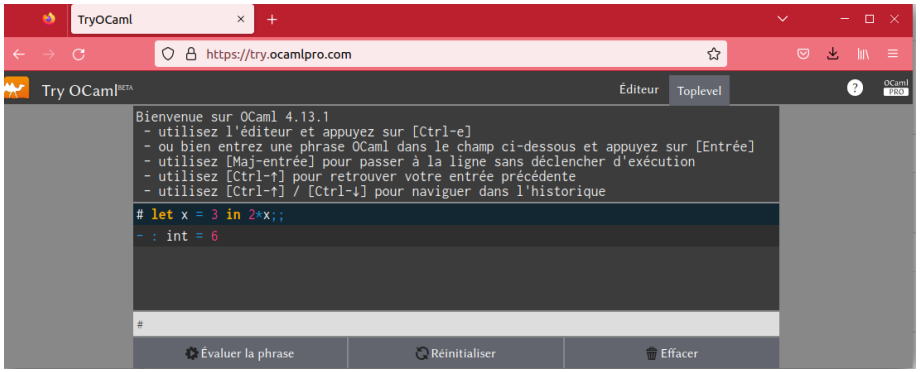
- TryOCaml ;



Interpréteurs en ligne

Si on n'a pas encore installé OCaml sur sa machine, on peut utiliser un interpréteur en ligne :

- TryOCaml ;

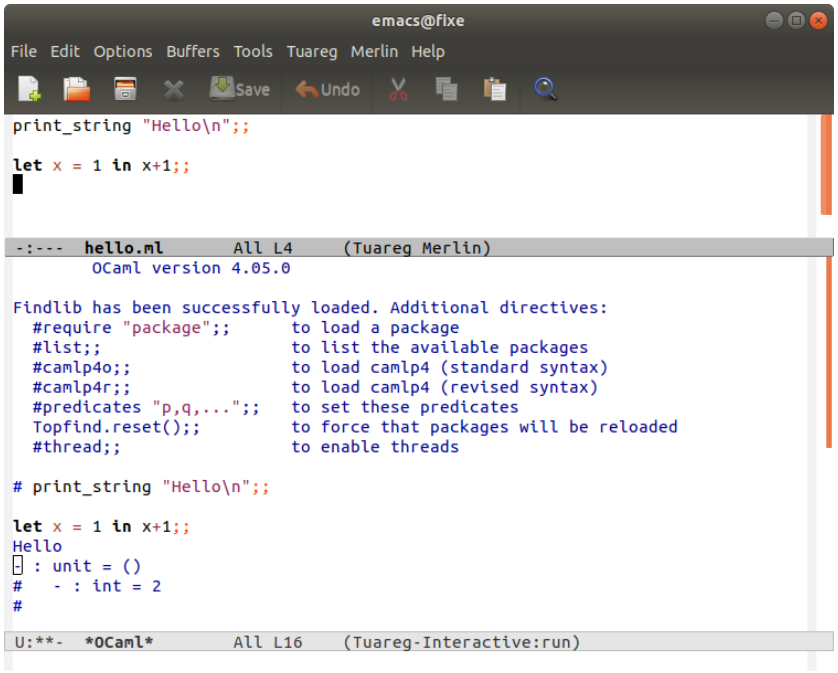


- Better OCaml ;

Raccourcis : `Ctrl+Enter` Exécute le code sélectionné ; ;`Ctrl+Shif+Enter` Exécute le code entier. ;`Ctrl+Space` montre le menu d'autocomplétion.

Tuareg

Pour ma part, j'utilise le greffon `Tuareg` d'`emacs` pour rendre plus agréable l'utilisation de l'interpréteur OCaml.



Consulter par exemple cette page.

Déclarations

- Syntaxe `let <id> = <expr>`. Le plus souvent `<id>` est un identificateur (il commence par une lettre minuscule ou un underscore). La partie à droite de l'égalité est une *expression* (elle peut être le résultat d'un calcul).

Déclarations

- Syntaxe `let <id> = <expr>`. Le plus souvent `<id>` est un identificateur (il commence par une lettre minuscule ou un underscore). La partie à droite de l'égalité est une *expression* (elle peut être le résultat d'un calcul).
- Une déclaration affecte le résultat de l'évaluation d'une expression à une variable, et est introduite par le mot clé `let`. Par exemple, écrivons ceci :

```
1 let x = 1 + 2;;
2 print_int x;;
3 let y = x * x;;
4 print_int y;;
```

Déclarations

- Syntaxe `let <id> = <expr>`. Le plus souvent `<id>` est un identificateur (il commence par une lettre minuscule ou un underscore). La partie à droite de l'égalité est une *expression* (elle peut être le résultat d'un calcul).
- Une déclaration affecte le résultat de l'évaluation d'une expression à une variable, et est introduite par le mot clé `let`. Par exemple, écrivons ceci :

```
1 let x = 1 + 2;;
2 print_int x;;
3 let y = x * x;;
4 print_int y;;
```

- Le programme calcule le résultat de `1+2`, l'affecte à la variable `x`, affiche la valeur de `x`, puis calcule le carré de `x`, l'affecte à la variable `y` et enfin affiche la valeur de `y`.

Déclarations

- Syntaxe `let <id> = <expr>`. Le plus souvent `<id>` est un identificateur (il commence par une lettre minuscule ou un underscore). La partie à droite de l'égalité est une *expression* (elle peut être le résultat d'un calcul).
- Une déclaration affecte le résultat de l'évaluation d'une expression à une variable, et est introduite par le mot clé `let`. Par exemple, écrivons ceci :

```
1 let x = 1 + 2;;
2 print_int x;;
3 let y = x * x;;
4 print_int y;;
```

- Le programme calcule le résultat de `1+2`, l'affecte à la variable `x`, affiche la valeur de `x`, puis calcule le carré de `x`, l'affecte à la variable `y` et enfin affiche la valeur de `y`.
- `x,y` sont des variables globales utilisables ailleurs dans le programme.

À propos des variables

- Une variable est nécessairement initialisée,

À propos des variables

- Une variable est nécessairement initialisée,
- Le type de la variable n'a pas besoin d'être déclaré, il est *inféré* par le compilateur.

À propos des variables

- Une variable est nécessairement initialisée,
- Le type de la variable n'a pas besoin d'être déclaré, il est *inféré* par le compilateur.
- Le contenu de la variable n'est pas modifiable. Dans le code précédent, `x` contient 3 pour toute la durée du programme. La variable est *immuable* ou *persistante* (on est en paradigme *fonctionnel*).

Variable persistante, vraiment ?

```
1 # let x =3;;
2 val x : int = 3
3 # let x = 4;;(*c'est une AUTRE variable x*)
4 val x : int = 4
```

Variable locale

- En C, les variables locales sont définies dans un bloc délimité par des accolades et elles n'ont pas d'existence hors de ce bloc.

```
1    {
2       int x = 10;
3       ...
4       }
5
```

Variable locale

- En C, les variables locales sont définies dans un bloc délimité par des accolades et elles n'ont pas d'existence hors de ce bloc.

```
1    {
2      int x = 10;
3      ...
4      }
5
```

- En OCaml, il n'y a pas de notion de bloc. Les variables locales d'une expression sont introduites par la construction `let in`.

```
1 # let x=10 in 2*x;;
2 - : int = 20
```

Variable locale

- En C, les variables locales sont définies dans un bloc délimité par des accolades et elles n'ont pas d'existence hors de ce bloc.

```
1    {
2      int x = 10;
3      ...
4      }
5
```

- En OCaml, il n'y a pas de notion de bloc. Les variables locales d'une expression sont introduites par la construction `let in`.

```
1 # let x=10 in 2*x;;
2 - : int = 20
```

- Les variables locales sont immuables et obligatoirement initialisées.

Variable locale

- Portée de la variable `x` :

```
# let x=10 in 2*x;;
- : int = 20
# x;;
Error: Unbound value x
```

Variable locale

- Portée de la variable `x` :

```
# let x=10 in 2*x;;
- : int = 20
# x;;
Error: Unbound value x
```

- Une déclaration locale est visible seulement dans l'expression qui suit le `in`.

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`
- caractères (`char`), `'a','b','1' ...` avec une seule quote.

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`
- caractères (`char`), `'a','b','1' ...` avec une seule quote.
- chaînes, (`string`), `"abc", "b"` (double quotes), concaténation :

```
# "toto" ^ " et gogo";;
- : string = "toto et gogo"
```

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`
- caractères (`char`), `'a','b','1' ...` avec une seule quote.
- chaînes, (`string`), `"abc", "b"` (double quotes), concaténation :

```
# "toto" ^ " et gogo";;
- : string = "toto et gogo"
```

- booléans, (`bool`), `true, false` ; opérateurs : `&&, || , not`

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`
- caractères (`char`), `'a','b','1' ...` avec une seule quote.
- chaînes, (`string`), `"abc", "b"` (double quotes), concaténation :

```
# "toto" ^ " et gogo";;
- : string = "toto et gogo"
```

- booléans, (`bool`), `true, false` ; opérateurs : `&&, || , not`
- Opérateur de comparaison de valeurs pour tous les types `=, >, <, >=, <=, <>`. Polymorphes, mais les deux arguments doivent avoir MEME type.

Types de base

- entiers (`int`), opérateurs : `+ - * / mod`
- flottants (`float`), opérateurs : `+. -. *. /.`
- caractères (`char`), `'a','b','1' ...` avec une seule quote.
- chaînes, (`string`), `"abc", "b"` (double quotes), concaténation :

```
# "toto" ^ " et gogo";;
- : string = "toto et gogo"
```

- booléans, (`bool`), `true, false` ; opérateurs : `&&, || , not`
- Opérateur de comparaison de valeurs pour tous les types `=, >, <, >=, <=, <>` . Polymorphes, mais les deux arguments doivent avoir MEME type.
- Opérateur de comparaison physique pour tous les types `==, !=`

Exemple d'opérations

- Quelques opérations :

```
# 1+2;;
- : int = 3
# 1.5+.3.5;;
- : float = 5.
# 2>7;;
- : bool = false
# "bonjour" > "bon";;
- : bool = true
# "durand" < "martin";;
- : bool = true
# "ab" = "ba";;
- : bool = false
# 6 mod 2;;
- : int = 0
```

Type `unit`

- Pour définir une fonction sans résultat, OCaml introduit le type `unit` qui ne contient qu'une seule valeur, notée `()`.

Type `unit`

- Pour définir une fonction sans résultat, OCaml introduit le type `unit` qui ne contient qu'une seule valeur, notée `()`.
- C'est par exemple le type de retour de la fonction `print_string`.

Type `unit`

- Pour définir une fonction sans résultat, OCaml introduit le type `unit` qui ne contient qu'une seule valeur, notée `()`.
- C'est par exemple le type de retour de la fonction `print_string`.
- Il sert aussi à définir des fonctions sans argument.

Type `unit`

- Pour définir une fonction sans résultat, OCaml introduit le type `unit` qui ne contient qu'une seule valeur, notée `()`.
- C'est par exemple le type de retour de la fonction `print_string`.
- Il sert aussi à définir des fonctions sans argument.
- Il joue un rôle comparable au type `void` de C.

Types construits predéfinis

- Les tuples (`t1 * t2 * .. * tn`) où `ti` est le type de la composante i du n-uplet.

```
# let x = false , 5, "hello";;
val x : bool * int * string = (false, 5, "hello")
# let a,b,c= x;; (*unpacking*);;
val a : bool = false
val b : int = 5
val c : string = "hello"
```

Types construits predéfinis

- Les tuples (`t1 * t2 * .. * tn`) où `ti` est le type de la composante i du n-uplet.

```
# let x = false , 5, "hello";;
val x : bool * int * string = (false, 5, "hello")
# let a,b,c= x;; (*unpacking*);;
val a : bool = false
val b : int = 5
val c : string = "hello"
```

- Les listes `t list`, où t est **le** type des éléments.

```
 # [1;2;3];; (* une liste d'entiers *)
- : int list = [1; 2; 3]
#  ["a";"bc"] @ ["bonjour"];;  (*@: operateur de
    concatenation*)
- : string list = ["a"; "bc"; "bonjour"]
# [1]@[];; (* []: liste vide  *)
- : int list = [1]
```

Plus sur les listes

- La liste vide est polymorphe.

```
1 # [];;(*la liste vide est polymorphe*)
2 - : 'a list = []
```

Plus sur les listes

- La liste vide est polymorphe.

```
# [];;(*la liste vide est polymorphe*)
- : 'a list = []
```

- Le module `List` contient des fonctions de manipulation de listes :

```
# List.length [1;2;3];; (* En O(n) !!! *)
- : int = 3
# List.hd [1;2;3];;(*tête (head) de la liste; O(1)*)
- : int = 1
# List.tl [1;2;3];;(*queue (tail) de la liste; O(1)*)
- : int list = [2; 3]
# 3::[2;1];; (*ajouter un élément en tête de liste*)
- : int list = [3; 2; 1]
# let a::b = [1;2;3];; (*séparer la tête de la queue*)
  ! Warning 8: this pattern-matching is not exhaustive !
val a : int = 1
val b : int list = [2; 3]
```

Plus sur les tuples

- Cas d'un tuple de taille 2 :

```
# let y = 1,6.2;;
val y : int * float = (1, 6.2)
# fst y;; (*pour first y*)
- : int = 1
# snd y;; (*pour second y*)
- : float = 6.2
```

Plus sur les tuples

- Cas d'un tuple de taille 2 :

```
# let y = 1,6.2;;
val y : int * float = (1, 6.2)
# fst y;; (*pour first y*)
- : int = 1
# snd y;; (*pour second y*)
- : float = 6.2
```

- Tuples contenant des tuples

```
# let z= (y,(2. +. 1.,4));;
val z : (int * float) * (float * int) = ((1, 6.2), (3.,
    4))
# let a,b = z;;
val a : int * float = (1, 6.2)
val b : float * int = (3., 4)
# let ((e,f),(g,h)) = z;;
val e : int = 1
val f : float = 6.2
val g : float = 3.
```

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

Le symbole `_` indique la présence d'une composante sans la nommer.

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

Le symbole `_` indique la présence d'une composante sans la nommer.

- Isomorphisme de types :

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

  Le symbole `_` indique la présence d'une composante sans la nommer.

- Isomorphisme de types :
  - Un élément de type `int * int * int` est un triplet d'entier.

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

Le symbole `_` indique la présence d'une composante sans la nommer.

- Isomorphisme de types :
  - Un élément de type `int * int * int` est un triplet d'entier.
  - Un élément de type `int * (int * int)` est un couple dont le premier élément est un entier et le second un couple d'entiers.

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

Le symbole `_` indique la présence d'une composante sans la nommer.

- Isomorphisme de types :
  - Un élément de type `int * int * int` est un triplet d'entier.
  - Un élément de type `int * (int * int)` est un couple dont le premier élément est un entier et le second un couple d'entiers.
  - Un élément de type `(int * int) * int` est un couple dont le premier élément est un couple d'entiers et le second un entier

Plus sur les tuples

- Déconstruire un tuple en ne prenant que certains items.

```
# let (_,(_,i)) = z;;
val i : int = 4
```

Le symbole `_` indique la présence d'une composante sans la nommer.

- Isomorphisme de types :
  - Un élément de type `int * int * int` est un triplet d'entier.
  - Un élément de type `int * (int * int)` est un couple dont le premier élément est un entier et le second un couple d'entiers.
  - Un élément de type `(int * int) * int` est un couple dont le premier élément est un couple d'entiers et le second un entier
  - Les 3 ensembles des éléments appartenant à chacun de ces 3 types sont en bijection canonique. C'est un exemple d'*isomorphisme de types*. Malheureusement, pour OCaml, ces types sont bien distincts !

Les mots clés `if then else`

- En OCaml, il n'y a pas de distinction expression/instruction : il n'y a que des expressions.

Les mots clés `if then else`

- En OCaml, il n'y a pas de distinction expression/instruction : il n'y a que des expressions.
- Les expressions conditionnelles sont définies à l'aide de l'opérateur ternaire `if e1 then e2 else e3`.

Les mots clés `if then else`

- En OCaml, il n'y a pas de distinction expression/instruction : il n'y a que des expressions.
- Les expressions conditionnelles sont définies à l'aide de l'opérateur ternaire `if e1 then e2 else e3`.
- L'expression `e1` doit renvoyer une valeur bouléenne.

Les mots clés `if then else`

- En OCaml, il n'y a pas de distinction expression/instruction : il n'y a que des expressions.
- Les expressions conditionnelles sont définies à l'aide de l'opérateur ternaire `if e1 then e2 else e3`.
- L'expression `e1` doit renvoyer une valeur bouléenne.
- En général (sauf cas particulier où `e2` est de type `unit`) la branche négative est obligatoire.

Les mots clés `if then else`

- En OCaml, il n'y a pas de distinction expression/instruction : il n'y a que des expressions.
- Les expressions conditionnelles sont définies à l'aide de l'opérateur ternaire `if e1 then e2 else e3`.
- L'expression `e1` doit renvoyer une valeur bouléenne.
- En général (sauf cas particulier où `e2` est de type `unit`) la branche négative est obligatoire.
- Une expression conditionnelle est polymorphe : *i.e.* `e2` peut être de n'importe quel type, mais `e3` est obligatoirement du même type que `e2`.

Exemple

```
# let x = 42;;
val x : int = 42
# let v = 10 + (if x > 0 then -1 else 4);;
val v : int = 9
```

- Branche négative non obligatoire si type `unit` :

```
# if (3 mod 2 = 1) then print_string "okayyyyy";;
okayyyyy- : unit = ()
```

Fonctions anonymes

- Les fonctions en OCaml sont des valeurs comme les autres. Syntaxe : `fun <id> -> <expr>` .

Fonctions anonymes

- Les fonctions en OCaml sont des valeurs comme les autres. Syntaxe : `fun <id> -> <expr>`.
  - `<id>` désigne le paramètre de la fonction. Il a une portée localisée au corps de la fonction.

Fonctions anonymes

- Les fonctions en OCaml sont des valeurs comme les autres. Syntaxe : `fun <id> -> <expr>`.
  - `<id>` désigne le paramètre de la fonction. Il a une portée localisée au corps de la fonction.
  - `->` désigne le début du corps de la fonction.

Fonctions anonymes

- Les fonctions en OCaml sont des valeurs comme les autres. Syntaxe : `fun <id> -> <expr>`.
  - `<id>` désigne le paramètre de la fonction. Il a une portée localisée au corps de la fonction.
  - `->` désigne le début du corps de la fonction.
  - `<expr>` est une expression quelconque qui peut utiliser `<id>`.

Exemples

- Fonction produit $x \mapsto x \times x$:

```
# fun x -> x*x;;(*fonction anonyme*)
- : int -> int = <fun>
# (fun x -> x*x) 5;;(*appliquer une fonction anonyme*)
- : int = 25
```

Exemples

- Fonction produit $x \mapsto x \times x$:

```
# fun x -> x*x;;(*fonction anonyme*)
- : int -> int = <fun>
# (fun x -> x*x) 5;;(*appliquer une fonction anonyme*)
- : int = 25
```

- Observer le type de la fonction. Il est noté sous la forme $\tau_1 \to \tau_2$.

Exemples

- Fonction produit $x \mapsto x \times x$:

```
# fun x -> x*x;;(*fonction anonyme*)
- : int -> int = <fun>
# (fun x -> x*x) 5;;(*appliquer une fonction anonyme*)
- : int = 25
```

- Observer le type de la fonction. Il est noté sous la forme $\tau_1 \to \tau_2$.
- Inférence de type : comme on utilise l'opérateur `*`, OCaml en déduit que le type de la fonction est `int -> int`.

Exemples

- Fonction produit $x \mapsto x \times x$:

```
# fun x -> x*x;;(*fonction anonyme*)
- : int -> int = <fun>
# (fun x -> x*x) 5;;(*appliquer une fonction anonyme*)
- : int = 25
```

- Observer le type de la fonction. Il est noté sous la forme $\tau_1 \to \tau_2$.
- Inférence de type : comme on utilise l'opérateur `*`, OCaml en déduit que le type de la fonction est `int -> int`.
- Pour nommer une fonction :

```
# let f = fun x y -> x +. y;;(*déclarer f*)
val f : float -> float -> float = <fun>
# f 1. 2.;;(*utiliser f*)
- : float = 3.
```

Sucre syntaxique ; Forcer le type du paramètre

- OCaml permet de déclarer une fonction de façon plus concise que `let f = fun ...`.

```
let f x y = x +. y;;
(*équivalent à let f = fun x y -> x+.y*)
f 1. 2.;;
```

Sucre syntaxique ; Forcer le type du paramètre

- OCaml permet de déclarer une fonction de façon plus concise que `let f = fun ...` .

```
1 let f x y = x +. y;;
2 (*équivalent à let f = fun x y -> x+.y*)
3 f 1. 2.;;
```

- On peut indiquer le type du paramètre et celui de retour :

```
1 let f (x:int) : int = x+x;;
```

C'est ici surtout utile pour le lecteur.

Sucre syntaxique ; Forcer le type du paramètre

- OCaml permet de déclarer une fonction de façon plus concise que `let f = fun ...` .

```
let f x y = x +. y;;
(*équivalent à let f = fun x y -> x+.y*)
f 1. 2.;;
```

- On peut indiquer le type du paramètre et celui de retour :

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

C'est ici surtout utile pour le lecteur.

- Attention aux priorités (penser aux parenthèses) :

```
# f 3 + 5, (f 3) + 5, f (3+5);;
- : int * int * int = (11, 11, 16)
```

En OCaml, l'application d'une fonction est syntaxiquement plus prioritaire que les autres opérations.

Application de fonction

- En programmation fonctionnelle, l'application de fonction est la seule opération qui permette d'effectuer un calcul. Même `2 + 3` est en fait du sucre syntaxique pour exprimer l'application de `(+)` à `2` et `3` : c.a.d `(+) 2 3`

Application de fonction

- En programmation fonctionnelle, l'application de fonction est la seule opération qui permette d'effectuer un calcul. Même `2 + 3` est en fait du sucre syntaxique pour exprimer l'application de `(+)` à `2` et `3` : c.a.d `(+) 2 3`
- Pour effectuer `<expr1> <expr 2>` :

Application de fonction

- En programmation fonctionnelle, l'application de fonction est la seule opération qui permette d'effectuer un calcul. Même `2 + 3` est en fait du sucre syntaxique pour exprimer l'application de `(+)` à `2` et `3` : c.a.d `(+) 2 3`
- Pour effectuer `<expr1> <expr 2>` :
  - On évalue `<expr1>`. C'est nécessairement une fonction de la forme `fun x -> e` de type $\tau \to \tau'$.

Application de fonction

- En programmation fonctionnelle, l'application de fonction est la seule opération qui permette d'effectuer un calcul. Même `2 + 3` est en fait du sucre syntaxique pour exprimer l'application de `(+)` à `2` et `3` : c.a.d `(+) 2 3`
- Pour effectuer `<expr1> <expr 2>` :
  - On évalue `<expr1>`. C'est nécessairement une fonction de la forme `fun x -> e` de type $\tau \to \tau'$.
  - On évalue `<expr 2>`. L'expression résultante `v` doit être de type τ.

Application de fonction

- En programmation fonctionnelle, l'application de fonction est la seule opération qui permette d'effectuer un calcul. Même `2 + 3` est en fait du sucre syntaxique pour exprimer l'application de `(+)` à `2` et `3` : c.a.d `(+) 2 3`
- Pour effectuer `<expr1> <expr 2>` :
  - On évalue `<expr1>`. C'est nécessairement une fonction de la forme `fun x -> e` de type $\tau \to \tau'$.
  - On évalue `<expr 2>`. L'expression résultante `v` doit être de type τ.
  - On évalue l'expression `e` dans un environnement où `x` est associé à la valeur `v`. La valeur renvoyée est de type τ'.

Fonction sans argument

- Une fonction sans argument prend en fait un unique argument de type `unit` (qu'on note `()`.

```
# let f () = print_string "coucou";;
val f : unit -> unit = <fun>
```

Fonction sans argument

- Une fonction sans argument prend en fait un unique argument de type `unit` (qu'on note `()`.

```
# let f () = print_string "coucou";;
val f : unit -> unit = <fun>
```

- Bien sûr, la fonction `f` ci-dessus peut être vue comme un alias pour l'expression `print_string "coucou"`.

Fonction sans argument

- Une fonction sans argument prend en fait un unique argument de type `unit` (qu'on note `()`).

```
# let f () = print_string "coucou";;
val f : unit -> unit = <fun>
```

- Bien sûr, la fonction `f` ci-dessus peut être vue comme un alias pour l'expression `print_string "coucou"`.
- Les fonctions dont le type de retour est `unit` sont souvent utilisées pour réaliser des affichages ou bien des mises à jour de variables mutables (tableaux, références...).

Fonction sans résultat

- Les fonctions sans résultat sont utilisées à des fins d'affichage et/ou pour réaliser des *effets de bord*.
  Pour le moment toutes nos variables sont persistantes mais on découvrira bientôt comment créer des variables mutables.

Fonction sans résultat

- Les fonctions sans résultat sont utilisées à des fins d'affichage et/ou pour réaliser des *effets de bord*.
  Pour le moment toutes nos variables sont persistantes mais on découvrira bientôt comment créer des variables mutables.
- Exemple de fonction sans résultat :

```
# let print_square x =
    Printf.printf "%f^0.5=%f" x (sqrt x);;
val print_square : float -> unit = <fun>
# print_square 2.;;(*Observer : '2.' pas '2'*)
2.000000^0.5=1.414214- : unit = ()
```

Ordre supérieur

- Un exemple avec passage de fonction en argument :

```
# let f x = x*x;;
val f : int -> int = <fun>
# let affiche f x = print_int (f x);;
val affiche : ('a -> int) -> 'a -> unit = <fun>
# affiche f 2;;
4- : unit = ()
```

Ordre supérieur

- Un exemple avec passage de fonction en argument :

```
# let f x = x*x;;
val f : int -> int = <fun>
# let affiche f x = print_int (f x);;
val affiche : ('a -> int) -> 'a -> unit = <fun>
# affiche f 2;;
4- : unit = ()
```

- Ci-dessus `affiche` prend deux arguments : une fonction d'un certain type (noté `'a`) vers les entiers, un élément de type `'a`. Le type de retour est `unit`. Le type de la fonction est donc `('a -> int) -> 'a -> unit = <fun>`

Ordre supérieur

- On a vu qu'on peut passer une fonction en argument. On peut aussi renvoyer une fonction.

Ordre supérieur

- On a vu qu'on peut passer une fonction en argument. On peut aussi renvoyer une fonction.
- Exemple : renvoie de l'homothétie de rapport x :

```
# let homothetie x = fun a -> a * x;;
val homothetie : int -> int -> int = <fun>
# (homothetie 3) 5;;
- : int = 15
# let g = homothetie 3 in g 5;;
- : int = 15
```

Application partielle

- Définissons une fonction à deux arguments :

```
# let milieu x y = (x+.y)/.2.;;
val milieu : float -> float -> float = <fun>
# milieu 2. 3.;;
- : float = 2.5
```

Application partielle

- Définissons une fonction à deux arguments :

```
# let milieu x y = (x+.y)/.2.;;
val milieu : float -> float -> float = <fun>
# milieu 2. 3.;;
- : float = 2.5
```

- Lorsqu'une fonction a plusieurs arguments, on n'est pas obligé de les lui fournir tous lors de l'application.
  Si on ne le fait pas le résultat de cette application partielle est lui même une fonction qui peut éventuellement être lié a un identificateur et être appliqué par la suite.

```
# let g = milieu 2.;;
val g : float -> float = <fun>
# g 3.;;
- : float = 2.5
```

Fonctions récursives

- Ecrivons la fonction `puissance`

```
1  let  puissance b n =
2         if n =0 then 1
3         else b * puissance b (n-1);;
4
5     Characters 69-78:
6         else b * puissance b (n-1);;
7                                ^^^^^^^^^
8  Error: Unbound value puissance
```

Fonctions récursives

- Ecrivons la fonction `puissance`

```
let  puissance b n =
       if n =0 then 1
       else b * puissance b (n-1);;

   Characters 69-78:
       else b * puissance b (n-1);;
                            ^^^^^^^^^
Error: Unbound value puissance
```

- Il faut prévenir le compilateur que la fonction est récursive en faisant suivre le mot clef `let` du mot clef `rec`.

```
let rec puissance b n =
       if n =0 then 1
       else b * puissance b (n-1);;
   val puissance : int -> int -> int = <fun>
```

Fonctions mutuellement récursives

- On utilise une construction de la forme

```
let rec g = ... and f = ...
```

Fonctions mutuellement récursives

- On utilise une construction de la forme
  `let rec g = ... and f = ...`
- Calcul de parité

```
# let rec pair n = (n=0) || impair (n-1)
and impair n =  (n <> 0) && pair (n-1);;
  val pair : int -> bool = <fun>
val impair : int -> bool = <fun>
# pair 5;;
- : bool = false
# impair 5;;
- : bool = true
```

Filtrage sur motif de l'argument

Quand on connaît la forme du motif de l'argument, on peut anticiper :

```
# let f ((x,(y,z)),t) = x*t-z*y;;
val f : (int * (int * int)) * int -> int = <fun>
# f ((1,(2,3)),4);;
- : int = -2
# f (1,2,3,4);;(*erreur de type*)
Characters 2-11:
  f (1,2,3,4);;
    ^^^^^^^^^
Error: This expression has type 'a * 'b * 'c * 'd
       but an expression was expected of type
       (int * (int * int)) * int
```

Émuler des boucles

- Pour le moment on ne présente pas les traits impératifs de OCaml.

Émuler des boucles

- Pour le moment on ne présente pas les traits impératifs de OCaml.
- Comment émuler une boucle comme celle-ci :

```
1 int x = ...;
2 while (x <45){
3   x = x+3
4 }
5
```

Émuler des boucles

- Pour le moment on ne présente pas les traits impératifs de OCaml.
- Comment émuler une boucle comme celle-ci :

```
1 int x = ...;
2 while (x <45){
3   x = x+3
4 }
5
```

- On peut le faire facilement avec une fonction comme celle-ci :

```
1 let rec loop x =
2   if x < 45 then loop (x+3) else x;;
```

Avertissement

- Les présents transparents ne sont qu'une introduction à OCaml.
- On ne présente ci-dessous que le cas particulier du filtrage sur les listes.

Listes

- Les listes d'objets de type τsont définies inductivement par

Listes

- Les listes d'objets de type τsont définies inductivement par
  - la liste vide `[]` est une liste d'objets de n'importe quel type donc en particulier de type τ ;

Listes

- Les listes d'objets de type τsont définies inductivement par
  - la liste vide `[]` est une liste d'objets de n'importe quel type donc en particulier de type τ ;
  - Si e est de type τ et si `t` est une liste de type τ, alors `e::t` est une liste d'objets de type τ.

Listes

- Les listes d'objets de type τsont définies inductivement par
  - la liste vide `[]` est une liste d'objets de n'importe quel type donc en particulier de type τ ;
  - Si e est de type τ et si `t` est une liste de type τ, alors `e::t` est une liste d'objets de type τ.
- On peut donc explorer une liste en la deconstruisant : on sépare son élément de tête du reste de la liste ; on traite l'élément de tête et on applique le même traitement au reste de la liste.

Conséquence de la définition inductive

- On a vu comment sont construites inductivement les listes. Il devient possible de les explorer.

Conséquence de la définition inductive

- On a vu comment sont construites inductivement les listes. Il devient possible de les explorer.
- Exemple calcul de la longueur :

```
# let rec longueur t = if t = [] then 0
      else let l = List.tl t in  1 + longueur l;;
val longueur : 'a list -> int = <fun>
# longueur [3;6;8];;
- : int = 3
```

Rappel : `List.tl t` donne la queue de liste (tout sauf le premier élément)

Conséquence de la définition inductive

- On a vu comment sont construites inductivement les listes. Il devient possible de les explorer.
- Exemple calcul de la longueur :

```
# let rec longueur t = if t = [] then 0
      else let l = List.tl t in  1 + longueur l;;
val longueur : 'a list -> int = <fun>
# longueur [3;6;8];;
- : int = 3
```

Rappel : `List.tl t` donne la queue de liste (tout sauf le premier élément)

- On pourrait aussi écrire, de façon plus proche de la définition inductive :

```
let rec longueur t = if t = [] then 0
      else let _::l =   in 1 + longueur l;;
```

Mais on recevrait un Warning pour risque de filtrage non exhaustif dans `let _::l = t`

Instruction de filtrage `match with`

- L'exemple simpliste du calcul de la longueur pourrait faire penser qu'on peut résoudre élégamment tous les problèmes sur les listes avec les opérateurs `if then else`.

Instruction de filtrage `match with`

- L'exemple simpliste du calcul de la longueur pourrait faire penser qu'on peut résoudre élégamment tous les problèmes sur les listes avec les opérateurs `if then else`.
- Mais on risque vite d'avoir une cascade inesthétique de `if then else`.

Instruction de filtrage `match with`

- L'exemple simpliste du calcul de la longueur pourrait faire penser qu'on peut résoudre élégamment tous les problèmes sur les listes avec les opérateurs `if then else`.
- Mais on risque vite d'avoir une cascade inesthétique de `if then else`.
- Pour éviter cela, l'instruction `match with` réalise un *filtrage* de motif.
  Elle permet soit de gérer différents cas en fonction des valeurs d'une expression, soit d'accéder aux éléments d'un type construit (ou les deux à la fois).

Instruction de filtrage `match with`

- L'exemple simpliste du calcul de la longueur pourrait faire penser qu'on peut résoudre élégamment tous les problèmes sur les listes avec les opérateurs `if then else`.
- Mais on risque vite d'avoir une cascade inesthétique de `if then else`.
- Pour éviter cela, l'instruction `match with` réalise un *filtrage* de motif.
  Elle permet soit de gérer différents cas en fonction des valeurs d'une expression, soit d'accéder aux éléments d'un type construit (ou les deux à la fois).
- Plus lisible que `if then else` en cascades `match with` effectue en outre une *analyse d'exhaustivité*. Il vérifie que tous les cas possibles ont bien été couverts ce qui est très utile pour le débugage.

Calcul de longueur avec filtrage

```
# let rec longueur l = match l with
  | [] -> 0
  | _::t -> 1+ longueur t;;
    val longueur : 'a list -> int = <fun>
# longueur [3;6;8];;
- : int = 3
```

Observer le `_::t` qui permet de ne pas tenir compte de la valeur de l'élément de tête, seulement de son existence.

Filtrage sur des tuples

- Le *filtrage* de motif permet soit de gérer différents cas en fonction des valeurs d'une expression, soit d'accéder aux éléments d'un type construit (ou les deux à la fois).

Filtrage sur des tuples

- Le *filtrage* de motif permet soit de gérer différents cas en fonction des valeurs d'une expression, soit d'accéder aux éléments d'un type construit (ou les deux à la fois).
- La fonction suivante filtre ses arguments (une paire) pour faire leur addition en tenant compte du cas où l'un d'entre eux est zéro :

```
let rec somme a b =
  match a,b with
  | 0,n -> n
  | n,0 -> n
  | _,_ (*autres cas*) -> 1 +
                          if a > b
                          then somme a (b-1)
                          else somme (a-1) b;;
val somme : int -> int -> int = <fun>
# somme 2 3;;(*exo : faire tourner à la main*)
- : int = 5
```