Bases de programmation en OCaml

Option informatique 2024-2025

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)
- Aujourd'hui présentation d'OCaml , même si option informatique $\neq$ programmer en OCaml

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)
- Aujourd'hui présentation d'OCaml , même si option informatique $\neq$ programmer en OCaml
- But de cette présentation

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)
- Aujourd'hui présentation d'OCaml , même si option informatique $\neq$ programmer en OCaml
- But de cette présentation
  - Présentation des grands traits de OCaml

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)
- Aujourd'hui présentation d'OCaml , même si option informatique $\neq$ programmer en OCaml
- But de cette présentation
  - Présentation des grands traits de OCaml
- vous devez apprendre deux langages différents (Python OCaml), sans les confondre,

Introduction

- Communication des documents exclusivement sur la plateforme Moodle (dès que possible...)
- Aujourd'hui présentation d'OCaml , même si option informatique $\neq$ programmer en OCaml
- But de cette présentation
  - Présentation des grands traits de OCaml
- vous devez apprendre deux langages différents (Python OCaml), sans les confondre,
- ni dans la syntaxe, ni dans leurs particularités.

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique
- Comment écrire des programmes en OCaml

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique
- Comment écrire des programmes en OCaml
  - Compilation soit pour machine virtuelle (pas ultra performant mais portable), soit pour machine dédiée (performant mais pas portable). On ne procédèrera pas ainsi, mais c'est très facile à faire

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique
- Comment écrire des programmes en OCaml
  - Compilation soit pour machine virtuelle (pas ultra performant mais portable), soit pour machine dédiée (performant mais pas portable). On ne procédèrera pas ainsi, mais c'est très facile à faire
  - Interpréteur : boucle interactive (TOPLEVEL), dans laquelle on écrit du code qui est interprété directement.

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique
- Comment écrire des programmes en OCaml
  - Compilation soit pour machine virtuelle (pas ultra performant mais portable), soit pour machine dédiée (performant mais pas portable). On ne procédèrera pas ainsi, mais c'est très facile à faire
  - Interpréteur : boucle interactive (TOPLEVEL), dans laquelle on écrit du code qui est interprété directement.
  - Au lycée environnement UBUNTU EMACS TUAREG-MODE (le tout éventuellement dans une machine virtuelle sous Windows)

OCaml , qu'est-ce que c'est ?

- Héritier de Caml (1987) développé par l'INRIA (Categorical Abstract Machine Language)
- OCaml = Objective Caml (1996)
- Ce n'est pas un langage gadget : industrie
- Assistant de preuve COQ : industrie aéronautique
- Comment écrire des programmes en OCaml
  - Compilation soit pour machine virtuelle (pas ultra performant mais portable), soit pour machine dédiée (performant mais pas portable). On ne procédèrera pas ainsi, mais c'est très facile à faire
  - Interpréteur : boucle interactive (TOPLEVEL), dans laquelle on écrit du code qui est interprété directement.
  - Au lycée environnement UBUNTU EMACS TUAREG-MODE (le tout éventuellement dans une machine virtuelle sous Windows)
  - Illustration

Grands traits de OCaml : langage fonctionnel

- Langage essentiellement fonctionnel : il traite les fonctions comme des valeurs comme les autres

Grands traits de OCaml : langage fonctionnel

- Langage essentiellement fonctionnel : il traite les fonctions comme des valeurs comme les autres
- Cf. maths avec des ensemble de fonctions. Une fonction est juste un élément d'un tel ensemble, comme 3 est un élément de $\mathbb{N}$.

Grands traits de OCaml : langage fonctionnel

- Langage essentiellement fonctionnel : il traite les fonctions comme des valeurs comme les autres
- Cf. maths avec des ensemble de fonctions. Une fonction est juste un élément d'un tel ensemble, comme 3 est un élément de $\mathbb{N}$.
- On commencera par cet aspect du langage, dans lequel une fonction peut être passée en paramètre ou même calculée par une autre fonction

Grands traits de OCaml : langage fonctionnel

- Langage essentiellement fonctionnel : il traite les fonctions comme des valeurs comme les autres
- Cf. maths avec des ensemble de fonctions. Une fonction est juste un élément d'un tel ensemble, comme 3 est un élément de $\mathbb{N}$.
- On commencera par cet aspect du langage, dans lequel une fonction peut être passée en paramètre ou même calculée par une autre fonction
- Dans l'approche fonctionnelle il n'y a pas de variable ni de boucles (inconditionnelle ou conditionnelle) ! Ça c'est du ressort des langages impératifs (comme Python)

Grands traits de OCaml : langage fonctionnel

- Langage essentiellement fonctionnel : il traite les fonctions comme des valeurs comme les autres
- Cf. maths avec des ensemble de fonctions. Une fonction est juste un élément d'un tel ensemble, comme 3 est un élément de $\mathbb{N}$.
- On commencera par cet aspect du langage, dans lequel une fonction peut être passée en paramètre ou même calculée par une autre fonction
- Dans l'approche fonctionnelle il n'y a pas de variable ni de boucles (inconditionnelle ou conditionnelle) ! Ça c'est du ressort des langages impératifs (comme Python)
- OCaml présente quand même des traits impératifs que nous étudierons plus tard.

Grands traits de OCaml : langage fortement typé

- toute expression a un type et OCaml vérifie à l'interprétation (ou compilation) qu'on utilise les opérateurs et les fonctions avec les bons types.

Grands traits de OCaml : langage fortement typé

- toute expression a un type et OCaml vérifie à l'interprétation (ou compilation) qu'on utilise les opérateurs et les fonctions avec les bons types.
- Cela introduit des contraintes fortes : une famille d'opérateurs pour les entiers (`+`, `-`, `*`, `/`, `mod`) et une autre pour les flottants (`+.`, `-.`, `*.`, `/.`, `**`)

Grands traits de OCaml : langage fortement typé

- toute expression a un type et OCaml vérifie à l'interprétation (ou compilation) qu'on utilise les opérateurs et les fonctions avec les bons types.
- Cela introduit des contraintes fortes : une famille d'opérateurs pour les entiers (`+`, `-`, `*`, `/`, `mod`) et une autre pour les flottants (`+.`, `-.`, `*.`, `/.`, `**`)
- Illustration

Grands traits de OCaml : langage fortement typé

- toute expression a un type et OCaml vérifie à l'interprétation (ou compilation) qu'on utilise les opérateurs et les fonctions avec les bons types.
- Cela introduit des contraintes fortes : une famille d'opérateurs pour les entiers (`+, -, *, /, mod`) et une autre pour les flottants (`+., -., *., /., **`)
- Illustration
- Pas de conversion implicite entre types, mais fonctions de conversion explicite : `int_of_float, string_of_bool,...`

Grands traits de OCaml : inférence du type des expressions

- Si on revient sur l'exemple précédent on voit que OCaml a déterminé seul le type des valeurs pour vérifier si le code était correct.

Grands traits de OCaml : inférence du type des expressions

- Si on revient sur l'exemple précédent on voit que OCaml a déterminé seul le type des valeurs pour vérifier si le code était correct.
- On reviendra là-dessus tout à l'heure lors de l'étude des fonctions

Grands traits de OCaml : inférence du type des expressions

- Si on revient sur l'exemple précédent on voit que OCaml a déterminé seul le type des valeurs pour vérifier si le code était correct.
- On reviendra là-dessus tout à l'heure lors de l'étude des fonctions
- Illustration

Grands traits de OCaml : typage statique

- Quand on définit une liaison (cf. infra), on ne peut plus la modifier. On ne peut que créer une nouvelle liaison.

Grands traits de OCaml : typage statique

- Quand on définit une liaison (cf. infra), on ne peut plus la modifier. On ne peut que créer une nouvelle liaison.
- Illustration

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase
- Une telle règle est par exemple "Sujet Verbe Complément."

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase
- Une telle règle est par exemple "Sujet Verbe Complément."
- Avec "Sujet = Article Nom"

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase
- Une telle règle est par exemple "Sujet Verbe Complément."
- Avec "Sujet = Article Nom"
- On peut alors écrire "La voiture est rouge."

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase
- Une telle règle est par exemple "Sujet Verbe Complément."
- Avec "Sujet = Article Nom"
- On peut alors écrire "La voiture est rouge."
- Mais on peut aussi écrire "Le route est diabétique." Même si cela n'a aucun sens (sauf en poésie éventuellement...)

Vocabulaire pour décrire les élements du langage

- Il faut faire preuve de beaucoup de rigueur pour ne pas écrire n'importe quoi (comme en math, physique etc...)
- En français on ne peut pas écrire une phrase comme "Rouge le ? voiture"
- C'est interdit par la **syntaxe** du français, i.e. l'ensemble des règles de constructions autorisées pour former une phrase
- Une telle règle est par exemple "Sujet Verbe Complément."
- Avec "Sujet = Article Nom"
- On peut alors écrire "La voiture est rouge."
- Mais on peut aussi écrire "Le route est diabétique." Même si cela n'a aucun sens (sauf en poésie éventuellement...)
- La question du sens d'une phrase relève de la **sémantique**.

Vocabulaire pour décrire les élements du langage

- En OCaml et dans tous les langages, c'est la même chose !

Vocabulaire pour décrire les élements du langage

- En OCaml et dans tous les langages, c'est la même chose !
- D'ailleurs en OCaml on écrit aussi des **phrases**, terminées syntaxiquement par `;;` , et l'interpréteur interprète ces phrases.

Vocabulaire pour décrire les élements du langage

- En OCaml et dans tous les langages, c'est la même chose !
- D'ailleurs en OCaml on écrit aussi des **phrases**, terminées syntaxiquement par `;;` , et l'interpréteur interprète ces phrases.
- L'ensemble des règles syntaxiques qui déterminent ce que l'on peut écrire est appelé **grammaire** du langage.

Vocabulaire pour décrire les élements du langage

- Exemple tiré de la documentation d'OCaml :

Vocabulaire pour décrire les élements du langage

- Exemple tiré de la documentation d'OCaml :

1.7 Expressions

```
expr ::= ident
       | valeur-globale
       | constante
       | ( expr )
       | begin expr end
       | ( expr : exprtype )
       | expr , expr { , expr}
       | constructeur-non-constant expr
       | expr :: expr
       | [ expr { ; expr} ]
       | [| expr { ; expr} |]
       | { étiquette = expr { ; étiquette = expr} }
       | expr expr
       | opérateur-préfixe expr
       | expr opérateur-infixe expr
       | expr . étiquette
       | expr . étiquette <- expr
       | expr .( expr )
       | expr .( expr ) <- expr
       | expr & expr
       | expr or expr
       | if expr then expr [else expr]
       | while expr do expr done
       | for ident = expr (to | downto) expr do expr done
       | expr ; expr
       | match expr with filtrage
       | function filtrage
```

Vocabulaire pour décrire les élements du langage

- Un des concepts centraux est celui **d'expression**

Vocabulaire pour décrire les élements du langage

- Un des concepts centraux est celui **d'expression**
- Une expression a un type et une valeur et est éventuellement liée à un identifiant.

Vocabulaire pour décrire les élements du langage

- Un des concepts centraux est celui **d'expression**
- Une expression a un type et une valeur et est éventuellement liée à un identifiant.
- Quand on demande à l'interpréteur d'évaluer une phrase il calcule l'expression correspondante et crée éventuellement une **liaison globale** d'un identifiant vers cette valeur

Vocabulaire pour décrire les élements du langage

- Un des concepts centraux est celui **d'expression**
- Une expression a un type et une valeur et est éventuellement liée à un identifiant.
- Quand on demande à l'interpréteur d'évaluer une phrase il calcule l'expression correspondante et crée éventuellement une **liaison globale** d'un identifiant vers cette valeur
- Exemple :

```
1 1 + 2 ;; (* Expression de type entier et
      de valeur 3 *)
```

Ceci **est** une expression.

```
1 let a = 1 + 2 ;; (* Liaison de l'
     identifiant a vers la valeur 3 *)
```

Cette phrase **n'est pas** une expression ! On ne peut pas substituer dans la grammaire une expression par ceci

```
1 (let a = 1 + 2) + 2 ;; (* Provoque une
     erreur de syntaxe *)
```

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type `int` ,

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type `int` ,
- **in**fixe car placé entre deux expressions entières

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type int ,
- **in**fixe car placé entre deux expressions entières
- On a vu déjà des opérateurs sur les entiers et les flottants de type float

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type int ,
- **in**fixe car placé entre deux expressions entières
- On a vu déjà des opérateurs sur les entiers et les flottants de type float
- Sur les booléens de type bool il y a le **et logique** `&&` , le **ou logique** `||` ainsi que l'opérateur **préfixe** `not`

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type int ,
- **in**fixe car placé entre deux expressions entières
- On a vu déjà des opérateurs sur les entiers et les flottants de type float
- Sur les booléens de type bool il y a le **et logique** `&&` , le **ou logique** `||` ainsi que l'opérateur **préfixe** `not`
- Sur les chaînes de caractères de type string il y a l'opérateur de concaténation `^` : `"bon" ^ "jour"`

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type int ,
- **in**fixe car placé entre deux expressions entières
- On a vu déjà des opérateurs sur les entiers et les flottants de type float
- Sur les booléens de type bool il y a le **et logique** `&&` , le **ou logique** `||` ainsi que l'opérateur **préfixe** `not`
- Sur les chaînes de caractères de type string il y a l'opérateur de concaténation `^` : `"bon" ^ "jour"`
- Il y a des opérateurs de comparaison `=, <, >, <=, >=`

Vocabulaire pour décrire les élements du langage

- Dans `1 + 2` , `+` est un **opérateur infixe**, qui agit sur 2 entiers de type int ,
- **in**fixe car placé entre deux expressions entières
- On a vu déjà des opérateurs sur les entiers et les flottants de type float
- Sur les booléens de type bool il y a le **et logique** `&&` , le **ou logique** `||` ainsi que l'opérateur **préfixe** `not`
- Sur les chaînes de caractères de type string il y a l'opérateur de concaténation `^` : `"bon" ^ "jour"`
- Il y a des opérateurs de comparaison `=, <, >, <=, >=`
- Signalons enfin le type caractère char : 'a' (quote de la touche 4).

Vocabulaire pour décrire les élements du langage

- En étudiant les listes, les tableaux on verra des **constructeurs**

Vocabulaire pour décrire les élements du langage

- En étudiant les listes, les tableaux on verra des **constructeurs**
- Par exemple le constructeur **cons**, noté `::` qui permet d'ajouter un élément en tête d'une liste `1 :: [2; 3; 4]`

Vocabulaire pour décrire les élements du langage

- En étudiant les listes, les tableaux on verra des **constructeurs**
- Par exemple le constructeur **cons**, noté `::` qui permet d'ajouter un élément en tête d'une liste `1 :: [2; 3; 4]`
- Enfin la notion de **motif** apparaîtra lors de la présentation du filtrage.

Vocabulaire pour décrire les élements du langage

- En étudiant les listes, les tableaux on verra des **constructeurs**
- Par exemple le constructeur **cons**, noté `::` qui permet d'ajouter un élément en tête d'une liste `1 :: [2; 3; 4]`
- Enfin la notion de **motif** apparaîtra lors de la présentation du filtrage.
- Il faudra être capable de bien catégoriser (expressions, opérateurs, constructeurs, motif...) les futurs éléments du langage que l'on présentera pour ne pas écrire n'importe quoi !

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.
- On peut cependant faire des liaisons **locales** dont la portée est limitée à la fin de la phrase où elle apparaît

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.
- On peut cependant faire des liaisons **locales** dont la portée est limitée à la fin de la phrase où elle apparaît
- Syntaxe `let ident = expr1 in expr2` :
  `let a = 3 in a * a;;`

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.
- On peut cependant faire des liaisons **locales** dont la portée est limitée à la fin de la phrase où elle apparaît
- Syntaxe `let ident = expr1 in expr2` :
  `let a = 3 in a * a;;`
- Illustration

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.
- On peut cependant faire des liaisons **locales** dont la portée est limitée à la fin de la phrase où elle apparaît
- Syntaxe `let ident = expr1 in expr2` :
  `let a = 3 in a * a;;`
- Illustration
- La totalité `let ident = expr1 in expr2` **est** une expression ayant pour valeur et type ceux de `expr2` .

Liaisons globales, locales, parallèles

- On vient de voir le mécanisme de liaison globale `let a = 3;;`
- L'identifiant `a` est ensuite connu et pourra être utilisé dans toutes les phrases suivantes. Sa **portée** (sa visibilité) est globale et s'étend à tout le code qui suit la définition.
- On peut cependant faire des liaisons **locales** dont la portée est limitée à la fin de la phrase où elle apparaît
- Syntaxe `let ident = expr1 in expr2` :
  `let a = 3 in a * a;;`
- Illustration
- La totalité `let ident = expr1 in expr2` **est** une expression ayant pour valeur et type ceux de `expr2` .
- On peut donc l'utiliser dans toute règle syntaxique utilisant une expression, et en particulier dans celle de déclaration d'une liaison locale : on peut imbriquer les liaisons locales

Liaisons globales, locales, parallèles

- Une liaison locale utilisant un identifiant défini dans une liaison précédente **MASQUE** temporairement cette liaison dans le code correspondant à sa portée.

Liaisons globales, locales, parallèles

- Une liaison locale utilisant un identifiant défini dans une liaison précédente **MASQUE** temporairement cette liaison dans le code correspondant à sa portée.
- Illustration

Liaisons globales, locales, parallèles

- Liaisons simultanées (en parallèle) avec le mot clé `and` , aussi bien dans les liaisons globales que locales.

Liaisons globales, locales, parallèles

- Liaisons simultanées (en parallèle) avec le mot clé `and` , aussi bien dans les liaisons globales que locales.
- Attention à ne pas utiliser `and` à la place de `&&` !

Liaisons globales, locales, parallèles

- Liaisons simultanées (en parallèle) avec le mot clé `and` , aussi bien dans les liaisons globales que locales.
- Attention à ne pas utiliser `and` à la place de `&&` !
- `let d = 1 and e = 2;;`

Liaisons globales, locales, parallèles

- Liaisons simultanées (en parallèle) avec le mot clé `and` , aussi bien dans les liaisons globales que locales.
- Attention à ne pas utiliser `and` à la place de `&&` !
- `let d = 1 and e = 2;;`
- `let b = 1 and c = 2 in b + c;;`

Liaisons globales, locales, parallèles

- Liaisons simultanées (en parallèle) avec le mot clé `and` , aussi bien dans les liaisons globales que locales.
- Attention à ne pas utiliser `and` à la place de `&&` !
- `let d = 1 and e = 2;;`
- `let b = 1 and c = 2 in b + c;;`
- Attention : les liaisons sont vraiment simultanées. On ne peut pas utiliser un identifiant qui ne serait déclaré qu'au même moment : `let e = 2 and f = e + 2;;`

Types

- On a déjà vu des types simples int , float , bool , char et string

Types

- On a déjà vu des types simples int , float , bool , char et string
- En OCaml on peut construire de nouveaux types, ce que l'on fera plus tard.

Types

- On a déjà vu des types simples int , float , bool , char et string
- En OCaml on peut construire de nouveaux types, ce que l'on fera plus tard.
- Mais on va déjà voir une première possibilité : le n-uplet ou tuple que l'on peut rapprocher de la notion de produit cartésien

Types

- On a déjà vu des types simples int , float , bool , char et string
- En OCaml on peut construire de nouveaux types, ce que l'on fera plus tard.
- Mais on va déjà voir une première possibilité : le n-uplet ou tuple que l'on peut rapprocher de la notion de produit cartésien
- Par exemple après `let x = (1, "Coucou", true)` `x` est de type int * string * bool (notez les étoiles entre les types)

Types

- On a déjà vu des types simples int , float , bool , char et string
- En OCaml on peut construire de nouveaux types, ce que l'on fera plus tard.
- Mais on va déjà voir une première possibilité : le n-uplet ou tuple que l'on peut rapprocher de la notion de produit cartésien
- Par exemple après `let x = (1, "Coucou", true)` `x` est de type int * string * bool (notez les étoiles entre les types)
- Déconstruction d'un n-uplet. Pour extraire les composantes du n-uplet on peut utiliser la syntaxe `let (a, b, c) = x` (éventuellement `in ...`).

Types

- Cas particulier des couples ou 2-uplets

Types

- Cas particulier des couples ou 2-uplets
- Il existe deux fonctions prédéfinies `fst` et `snd` (pour first et second) qui rendent respectivement la première et la seconde composante du couple.

Types

- Cas particulier des couples ou 2-uplets
- Il existe deux fonctions prédéfinies `fst` et `snd` (pour first et second) qui rendent respectivement la première et la seconde composante du couple.
- Cela ne fonctionne que pour les couples !

Types

- Cas particulier des couples ou 2-uplets
- Il existe deux fonctions prédéfinies `fst` et `snd` (pour first et second) qui rendent respectivement la première et la seconde composante du couple.
- Cela ne fonctionne que pour les couples !
- Illustration

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...
- Utilisation `f 3;;` . On peut écrire `f (3);;` mais ce n'est pas fait en pratique. Les parenthèses sont inutiles ici.

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...
- Utilisation `f 3;;` . On peut écrire `f (3);;` mais ce n'est pas fait en pratique. Les parenthèses sont inutiles ici.
- Attention les appels de fonction sont associés à gauche, i.e.

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...
- Utilisation `f 3;;` . On peut écrire `f (3);;` mais ce n'est pas fait en pratique. Les parenthèses sont inutiles ici.
- Attention les appels de fonction sont associés à gauche, i.e.
- `f g h i j x` est interprété comme `((((f g) h) i) j) x)` . Il y a intérêt à ce que les types soient compatibles...

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...
- Utilisation `f 3;;` . On peut écrire `f (3);;` mais ce n'est pas fait en pratique. Les parenthèses sont inutiles ici.
- Attention les appels de fonction sont associés à gauche, i.e.
- `f g h i j x` est interprété comme `((((f g) h) i) j) x)` . Il y a intérêt à ce que les types soient compatibles...
- pour calculer $f(f(3))$ il faut écrire `f (f 3);;`

Fonctions : à un seul paramètre

- Il faut bien sûr pouvoir écrire des fonctions et pouvoir les appliquer à des paramètres. On va distinguer les fonctions à 1 paramètre et les fonctions à plusieurs paramètres
- Pour fonction à un paramètre
- Déclaration : `let f x = expr`
- Par exemple `let f x = x + 1;;`
- val f : int -> int = <fun>
- Ce qui montre qu'une fonction est une valeur comme une autre...
- Utilisation `f 3;;` . On peut écrire `f (3);;` mais ce n'est pas fait en pratique. Les parenthèses sont inutiles ici.
- Attention les appels de fonction sont associés à gauche, i.e.
- `f g h i j x` est interprété comme `((((f g) h) i) j) x)` . Il y a intérêt à ce que les types soient compatibles...
- pour calculer $f(f(3))$ il faut écrire `f (f 3);;`
- `f f 3;;` provoque une erreur de typage

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`
- En cas de doute : parenthésez vos expressions !

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`
- En cas de doute : parenthésez vos expressions !
- Notez qu'il n'y a pas de **return** dans le code de la fonction ! Lors de l'application de la fonction l'interpréteur va évaluer l'expression après le `=` , et la valeur obtenue est celle de l'application de la fonction

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`
- En cas de doute : parenthésez vos expressions !
- Notez qu'il n'y a pas de **return** dans le code de la fonction ! Lors de l'application de la fonction l'interpréteur va évaluer l'expression après le `=` , et la valeur obtenue est celle de l'application de la fonction
- Enfin on va se forcer à donner des informations supplémentaires (pas obligatoires) mais qui sont un garde-fou pour éviter de se tromper de type :

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`
- En cas de doute : parenthésez vos expressions !
- Notez qu'il n'y a pas de **return** dans le code de la fonction ! Lors de l'application de la fonction l'interpréteur va évaluer l'expression après le `=` , et la valeur obtenue est celle de l'application de la fonction
- Enfin on va se forcer à donner des informations supplémentaires (pas obligatoires) mais qui sont un garde-fou pour éviter de se tromper de type :
- On ajoute dans la déclaration le type du paramètre et celui de l'expression calculée, avec la syntaxe suivante

Fonctions : à un seul paramètre

- Les opérateurs algébriques gardent cependant la préséance
- pour calculer $f(3) + f(3)$ on peut écrire `f 3 + f 3;;`
- En cas de doute : parenthésez vos expressions !
- Notez qu'il n'y a pas de **return** dans le code de la fonction ! Lors de l'application de la fonction l'interpréteur va évaluer l'expression après le `=` , et la valeur obtenue est celle de l'application de la fonction
- Enfin on va se forcer à donner des informations supplémentaires (pas obligatoires) mais qui sont un garde-fou pour éviter de se tromper de type :
- On ajoute dans la déclaration le type du paramètre et celui de l'expression calculée, avec la syntaxe suivante
- `let f (x : int) : int = x + 1;;`

Fonctions : anonyme à un seul paramètre

- On peut créer une fonction à un seul paramètre sans nom par le mot clé `function` et la syntaxe `function ident -> expr` (Notez le `->` très proche de la notation mathématique).

Fonctions : anonyme à un seul paramètre

- On peut créer une fonction à un seul paramètre sans nom par le mot clé `function` et la syntaxe `function ident -> expr` (Notez le `->` très proche de la notation mathématique).
- `function x -> x + 1;;` est une fonction avec la même fonctionnalité que f, mais sans nom !

Fonctions : anonyme à un seul paramètre

- On peut créer une fonction à un seul paramètre sans nom par le mot clé `function` et la syntaxe `function ident -> expr` (Notez le `->` très proche de la notation mathématique).
- `function x -> x + 1;;` est une fonction avec la même fonctionnalité que f, mais sans nom !
- Peut être utile dans certains cas où on ne veut pas créer une liaison vers la fonction, mais simplement la passer en paramètre

Fonctions : anonyme à un seul paramètre

- On peut créer une fonction à un seul paramètre sans nom par le mot clé `function` et la syntaxe `function ident -> expr` (Notez le `->` très proche de la notation mathématique).
- `function x -> x + 1;;` est une fonction avec la même fonctionnalité que f, mais sans nom !
- Peut être utile dans certains cas où on ne veut pas créer une liaison vers la fonction, mais simplement la passer en paramètre
- On peut bien sûr aussi lier la fonction à un identifiant (et ainsi lui redonner un nom) avec `let` : `let f = function x -> x + 1;;`

Fonctions : plusieurs paramètres

- Plusieurs solutions

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).
- Autre solution séparer les arguments par un simple espace

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).
- Autre solution séparer les arguments par un simple espace
- `let f2 x y z = x + y + z;;`

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).
- Autre solution séparer les arguments par un simple espace
- `let f2 x y z = x + y + z;;`
- crée une fonction de type int -> int -> int -> int

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).
- Autre solution séparer les arguments par un simple espace
- `let f2 x y z = x + y + z;;`
- crée une fonction de type int -> int -> int -> int
- Noter la différence de type qu'il faut savoir interpréter

Fonctions : plusieurs paramètres

- Plusieurs solutions
- En faire une fonction à une seule variable qui soit un n-uplet.
- Par exemple `let f1 (x, y, z) = x + y + z;;`
- crée une fonction de type int * int * int -> int
- C'est bien une fonction à un seul paramètre de type int * int * int et qui calcule un entier.
- Cette forme est dite **non curryfiée** (explication après).
- Autre solution séparer les arguments par un simple espace
- `let f2 x y z = x + y + z;;`
- crée une fonction de type int -> int -> int -> int
- Noter la différence de type qu'il faut savoir interpréter
- C'est la forme **curryfiée** (Haskell Curry).

Fonctions : plusieurs paramètres

- Le type int -> int -> int -> int nous dit que `f2` est une fonction qui à un entier associe une fonction

Fonctions : plusieurs paramètres

- Le type int -> int -> int -> int nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !

Fonctions : plusieurs paramètres

- Le type int -> int -> int -> int nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !

Fonctions : plusieurs paramètres

- Le type int -> int -> int -> int nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !
- Ainsi `f2 3;;` est la fonction : $y, z \rightarrow 3 + y + z$

Fonctions : plusieurs paramètres

- Le type int -> int -> int -> int nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !
- Ainsi `f2 3;;` est la fonction : $y, z \rightarrow 3 + y + z$
- C'est ce qu'on appelle une fonction **partielle**.

Fonctions : plusieurs paramètres

- Le type `int -> int -> int -> int` nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !
- Ainsi `f2 3;;` est la fonction : $y, z \rightarrow 3 + y + z$
- C'est ce qu'on appelle une fonction **partielle**.
- Et de même `f2 3 1;;` est la fonction : $z \rightarrow 4 + z$

Fonctions : plusieurs paramètres

- Le type `int -> int -> int -> int` nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !
- Ainsi `f2 3;;` est la fonction : $y, z \rightarrow 3 + y + z$
- C'est ce qu'on appelle une fonction **partielle**.
- Et de même `f2 3 1;;` est la fonction : $z \rightarrow 4 + z$
- Pour les fonctions à plusieurs variables on utilise `fun` (en forme curryfiée forcément...)

Fonctions : plusieurs paramètres

- Le type `int -> int -> int -> int` nous dit que `f2` est une fonction qui à un entier associe une fonction
- qui à un entier associe une fonction !
- qui à un entier associe un entier !
- Ainsi `f2 3;;` est la fonction : $y, z \rightarrow 3 + y + z$
- C'est ce qu'on appelle une fonction **partielle**.
- Et de même `f2 3 1;;` est la fonction : $z \rightarrow 4 + z$
- Pour les fonctions à plusieurs variables on utilise `fun` (en forme curryfiée forcément...)
- `fun x y z-> x + y + z;;` a la même fonctionnalité que `f2` mais sans nom !

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres
- typées

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres
- typées
- Peuvent être calculées, passées en paramètre, renvoyées par une fonction...

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres
- typées
- Peuvent être calculées, passées en paramètre, renvoyées par une fonction...
- typage associatif à droite, c'est-à-dire que

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres
- typées
- Peuvent être calculées, passées en paramètre, renvoyées par une fonction...
- typage associatif à droite, c'est-à-dire que
- `int -> (int -> (int -> int))` est équivalent à

Fonctions : plusieurs paramètres

- Les fonctions sont des valeurs comme les autres
- typées
- Peuvent être calculées, passées en paramètre, renvoyées par une fonction...
- typage associatif à droite, c'est-à-dire que
- int -> (int -> (int -> int)) est équivalent à
- int -> int -> int -> int . CAML a fait et fera la simplification d'ailleurs

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?
- Il y aura une flèche car c'est une fonction

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?
- Il y aura une flèche car c'est une fonction
- Mais cette fonction peut s'appliquer à n'importe quel couple, aussi bien int * int que char * bool !

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?
- Il y aura une flèche car c'est une fonction
- Mais cette fonction peut s'appliquer à n'importe quel couple, aussi bien int * int que char * bool !
- Regardons ce que rend OCaml

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?
- Il y aura une flèche car c'est une fonction
- Mais cette fonction peut s'appliquer à n'importe quel couple, aussi bien int * int que char * bool !
- Regardons ce que rend OCaml
- Caml utilise alors un type inconnu 'a , puis 'b s'il y en a besoin d'un autre...etc.. Notez bien l'apostrophe qui précède le nom du type

Fonctions : plusieurs paramètres

- Revenons sur les deux fonctions `fst` et `snd` qui rendent respectivement le premier élément et le second élément d'un couple (2-uplet)
- Que va rendre `fst` ?
- Il y aura une flèche car c'est une fonction
- Mais cette fonction peut s'appliquer à n'importe quel couple, aussi bien int * int que char * bool !
- Regardons ce que rend OCaml
- Caml utilise alors un type inconnu 'a , puis 'b s'il y en a besoin d'un autre...etc.. Notez bien l'apostrophe qui précède le nom du type
- Ces fonctions sont dites **polymorphes** ;

Fonctions : récursivité

- Dans le cadre de la programmation fonctionnelle il est indispensable qu'une fonction puisse s'appeler elle-même (rappel : pas de boucle par exemple).

Fonctions : récursivité

- Dans le cadre de la programmation fonctionnelle il est indispensable qu'une fonction puisse s'appeler elle-même (rappel : pas de boucle par exemple).
- C'est la récursivité déjà vue en S1 d'ITC

Fonctions : récursivité

- Dans le cadre de la programmation fonctionnelle il est indispensable qu'une fonction puisse s'appeler elle-même (rappel : pas de boucle par exemple).
- C'est la récursivité déjà vue en S1 d'ITC
- Pour qu'une fonction puisse s'appeler elle-même il faut utiliser explicitement (contraitement à Python) le mot clé `rec` après le `let` selon la syntaxe `let rec f x = ...`

Fonctions : récursivité

- Dans le cadre de la programmation fonctionnelle il est indispensable qu'une fonction puisse s'appeler elle-même (rappel : pas de boucle par exemple).
- C'est la récursivité déjà vue en S1 d'ITC
- Pour qu'une fonction puisse s'appeler elle-même il faut utiliser explicitement (contraitement à Python) le mot clé `rec` après le `let` selon la syntaxe `let rec f x = ...`
- On peut définir plusieurs fonctions mutuellement récursives avec le mot clé `and` :

```
1 # let rec f x = ...
2   and g x = ... ;;
```

Fonctions : récursivité

- Dans le cadre de la programmation fonctionnelle il est indispensable qu'une fonction puisse s'appeler elle-même (rappel : pas de boucle par exemple).
- C'est la récursivité déjà vue en S1 d'ITC
- Pour qu'une fonction puisse s'appeler elle-même il faut utiliser explicitement (contraitement à Python) le mot clé `rec` après le `let` selon la syntaxe `let rec f x = ...`
- On peut définir plusieurs fonctions mutuellement récursives avec le mot clé `and` :

```
# let rec f x = ...
  and g x = ... ;;
```

- Pour faciliter la programmation récursive, OCaml propose un trait particulièrement puissant : le filtrage par motif ou pattern matching.

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !
- une première syntaxe :

```
1 match expr0 with
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !
- une première syntaxe :

```
1 match expr0 with
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

- L'expression `expr0` est évaluée, et sa valeur comparée au premier motif :

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !
- une première syntaxe :

```
1 match expr0 with
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

- L'expression `expr0` est évaluée, et sa valeur comparée au premier motif :
  - si cette valeur satisfait les contraintes structurelles du motif, c'est `expr1` qui sera évaluée en réponse au filtrage ;

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !
- une première syntaxe :

```
1 match expr0 with
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

- L'expression `expr0` est évaluée, et sa valeur comparée au premier motif :
  - si cette valeur satisfait les contraintes structurelles du motif, c'est `expr1` qui sera évaluée en réponse au filtrage ;
  - sinon, sa valeur est comparée au second motif, et ainsi de suite...

Filtrage par motif (pattern matching)

- C'est un trait extrêmement puissant de CAML ;
- qui permet de faire des calculs (et définir des fonctions) par cas distincts ; C'est une expression !
- une première syntaxe :

```
1 match expr0 with
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

- L'expression `expr0` est évaluée, et sa valeur comparée au premier motif :
  - si cette valeur satisfait les contraintes structurelles du motif, c'est `expr1` qui sera évaluée en réponse au filtrage ;
  - sinon, sa valeur est comparée au second motif, et ainsi de suite...
  - Si aucun motif n'est reconnu, une exception `Match_failure` est levée.

Filtrage par motif (pattern matching)

- La totalité (du `match` au `exprn`) est une expression, dont le type et la valeur sont ceux de l'expression qui sera évaluée.

Filtrage par motif (pattern matching)

- La totalité (du `match` au `exprn`) est une expression, dont le type et la valeur sont ceux de l'expression qui sera évaluée.
- Cela suppose que toutes les expressions `expr1` à `exprn` aient le même type...

Filtrage par motif (pattern matching)

- Une autre syntaxe possible avec `function`

```
1 let f = function
2 | motif1 -> expr1
3 | motif2 -> expr2
4 | ..............
5 | motifn -> exprn ;;
```

Filtrage par motif (pattern matching)

- Une autre syntaxe possible avec `function`

```
let f = function
| motif1 -> expr1
| motif2 -> expr2
| ..............
| motifn -> exprn ;;
```

- Par exemple

```
# let sinc = function
    | 0. -> 1.
    | x -> sin(x) /. x ;;
val sinc : float -> float = <fun>
```

Filtrage par motif (pattern matching)

- Une autre syntaxe possible avec `function`

```
let f = function
| motif1 -> expr1
| motif2 -> expr2
| ..............
| motifn -> exprn ;;
```

- Par exemple

```
# let sinc = function
    | 0. -> 1.
    | x -> sin(x) /. x ;;
val sinc : float -> float = <fun>
```

- Noter que l'argument de la fonction n'est pas nommé juste après `function` , mais qu'on peut utiliser le nom du motif pour le calcul. On peut penser qu'il y a en plus un `match x with` où `x` est le paramètre sur lequel est appliquée la fonction

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.
- On peut mettre des identificateurs. Il est **HAUTEMENT CONSEILLÉ** de ne pas réutiliser un identificateur déjà défini.

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.
- On peut mettre des identificateurs. Il est **HAUTEMENT CONSEILLÉ** de ne pas réutiliser un identificateur déjà défini.
- Un identificateur ne peut apparaître **QU'UNE SEULE FOIS** dans un motif.

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.
- On peut mettre des identificateurs. Il est **HAUTEMENT CONSEILLÉ** de ne pas réutiliser un identificateur déjà défini.
- Un identificateur ne peut apparaître **QU'UNE SEULE FOIS** dans un motif.
- On peut mettre des constantes (en fait des **CONSTRUCTEURS CONSTANTS**).

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.
- On peut mettre des identificateurs. Il est **HAUTEMENT CONSEILLÉ** de ne pas réutiliser un identificateur déjà défini.
- Un identificateur ne peut apparaître **QU'UNE SEULE FOIS** dans un motif.
- On peut mettre des constantes (en fait des **CONSTRUCTEURS CONSTANTS**).
- ou des **CONSTRUCTEURS NON CONSTANTS**

Filtrage par motif (pattern matching)

- Que peut-on mettre dans un motif ?
- Un motif c'est une contrainte structurelle, mais pas une contrainte de calcul.
- On peut mettre des identificateurs. Il est **HAUTEMENT CONSEILLÉ** de ne pas réutiliser un identificateur déjà défini.
- Un identificateur ne peut apparaître **QU'UNE SEULE FOIS** dans un motif.
- On peut mettre des constantes (en fait des **CONSTRUCTEURS CONSTANTS**).
- ou des **CONSTRUCTEURS NON CONSTANTS**
- **MAIS PAS D'OPÉRATEURS**. On ne peut pas faire de calculs dans un motif.

Filtrage par motif (pattern matching)

- Il existe en plus un motif particulier _ (sous la touche 8) qui correspond à toute valeur :

Filtrage par motif (pattern matching)

- Il existe en plus un motif particulier _ (sous la touche 8) qui correspond à toute valeur :
- Par exemple pour écrire la fonction factorielle récursivement :

```
# let rec fact n =
    match n with
    | 0 -> 1
    | _ -> n * fact (n - 1) ;;
val fact : int -> int = <fun>
```

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.
- On ne peut pas écrire

```
# let est_pair n = match n with
    | 2 * p -> true
    | _ -> false ;;
```

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.
- On ne peut pas écrire

```
# let est_pair n = match n with
    | 2 * p -> true
    | _ -> false ;;
```

- On peut alors utiliser une condition supplémentaire : une **garde** selon la syntaxe

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.
- On ne peut pas écrire

```
1 # let est_pair n = match n with
2       | 2 * p -> true
3       | _ -> false ;;
```

- On peut alors utiliser une condition supplémentaire : une **garde** selon la syntaxe
- `| motif when condition -> expr` où condition est une expression de type bool

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.
- On ne peut pas écrire

```
1 # let est_pair n = match n with
2       | 2 * p -> true
3       | _ -> false ;;
```

- On peut alors utiliser une condition supplémentaire : une **garde** selon la syntaxe
- `| motif when condition -> expr` où condition est une expression de type bool
- L'expression ne sera évaluée que si le motif est satisfait et que la condition est évaluée à `true`

Filtrage gardé

- On ne peut pas forcément tout contraindre par un motif.
- On ne peut pas écrire

```
# let est_pair n = match n with
    | 2 * p -> true
    | _ -> false ;;
```

- On peut alors utiliser une condition supplémentaire : une **garde** selon la syntaxe
- `| motif when condition -> expr` où condition est une expression de type bool
- L'expression ne sera évaluée que si le motif est satisfait et que la condition est évaluée à `true`
- Par exemple on pourra écrire

```
# let est_pair n = match n with
    | m when m mod 2 = 0 -> true
    | _ -> false ;;
est_pair : int -> bool = <fun>
```

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=`, `<`, `+`, `+.`, `*`, `*.`, `&&`, `||`, `^`) : `a = b`

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=`, `<`, `+`, `+.`, `*`, `*.`, `&&`, `||`, `^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=`, `<`, `+`, `+.`, `*`, `*.`, `&&`, `||`, `^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=`, `<`, `+`, `+.`, `*`, `*.`, `&&`, `||`, `^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=`, `<`, `+`, `+.`, `*`, `*.`, `&&`, `||`, `^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;
- Utiliser la possibilité de commenter le code ; `(* mon commentaire *)` en CAML ;

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=, <, +, +., *, *., &&, ||, ^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;
- Utiliser la possibilité de commenter le code ; `(* mon commentaire *)` en CAML ;
- Indenter le texte pour la lisibilité (**indentation non significative**) ;

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=, <, +, +., *, *., &&, ||, ^`) : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;
- Utiliser la possibilité de commenter le code ; `(* mon commentaire *)` en CAML ;
- Indenter le texte pour la lisibilité (**indentation non significative**) ;
- Tout en minuscules (sauf exception) et sans accent ;

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=, <, +, +., *, *., &&, ||, ^ )` : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;
- Utiliser la possibilité de commenter le code ; `(* mon commentaire *)` en CAML ;
- Indenter le texte pour la lisibilité (**indentation non significative**) ;
- Tout en minuscules (sauf exception) et sans accent ;
- En devoir (et surtout au concours), votre code avec commentaires et explications sera lu 3 fois au maximum

Recommandations d'écriture

- Mettre un espace de part et d'autre de tout **opérateur** (`=, <, +, +., *, *., &&, ||, ^ )` : `a = b`
- Coller la virgule ou le point virgule au terme précédent, et mettre un espace après : `[1; 2; 3]`
- Coller le terme après la parenthèse/crochet ouvrante et la parenthèse/crochet fermante au terme précédent : `(1, 2, 3)`
- Utiliser des noms de variables parlants ;
- Utiliser la possibilité de commenter le code ; `(* mon commentaire *)` en CAML ;
- Indenter le texte pour la lisibilité (**indentation non significative**) ;
- Tout en minuscules (sauf exception) et sans accent ;
- En devoir (et surtout au concours), votre code avec commentaires et explications sera lu 3 fois au maximum
- Si le correcteur ne comprend toujours pas, il passe à la question suivante...

Gestion des erreurs

- À notre niveau si on veut signaler une erreur dans l'utilisation d'une fonction

Gestion des erreurs

- À notre niveau si on veut signaler une erreur dans l'utilisation d'une fonction
- on lèvera une exception avec l'instruction `failwith`

```
# let divise x y = match y with
    | 0. -> failwith "Division par zéro interdite !"
    | _ -> x /. y ;;
val divise : float -> float -> float
```

Gestion des erreurs

- À notre niveau si on veut signaler une erreur dans l'utilisation d'une fonction
- on lèvera une exception avec l'instruction `failwith`

```
# let divise x y = match y with
    | 0. -> failwith "Division par zéro interdite !"
    | _ -> x /. y ;;
val divise : float -> float -> float
```

- On apprendra plus tard à définir nos propres exceptions.

Résumé

- Liaison globale vs liaison locale

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.
  - `int, float, bool, char, string, unit, ...`

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.
  - `int, float, bool, char, string, unit, ...`
  - Pas de conversion **implicite** de types comme en python. On utilise **explicitement** `int_of_float` , etc...

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.
  - `int, float, bool, char, string, unit, ...`
  - Pas de conversion **implicite** de types comme en python. On utilise **explicitement** `int_of_float` , etc...
  - Les calculs se font à l'aide d'opérateurs spécifiques à chaque type `+, +., *, *., &&, ||, ^`

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.
  - `int, float, bool, char, string, unit, ...`
  - Pas de conversion **implicite** de types comme en python. On utilise **explicitement** `int_of_float` , etc...
  - Les calculs se font à l'aide d'opérateurs spécifiques à chaque type `+, +., *, *., &&, ||, ^`
  - exceptions : certains opérateurs sont polymorphes `<, <=` etc...

Résumé

- Liaison globale vs liaison locale
  - statiques toutes les deux
  - portée : tout ce qui suit la déclaration pour liaison globale, seulement dans le `in` pour liaison locale
  - liaison locale est une **expression** (valeur, type), pas la liaison globale.
- OCaml est un langage **fortement typé**.
  - `int, float, bool, char, string, unit, ...`
  - Pas de conversion **implicite** de types comme en python. On utilise **explicitement** `int_of_float` , etc...
  - Les calculs se font à l'aide d'opérateurs spécifiques à chaque type `+, +., *, *., &&, ||, ^`
  - exceptions : certains opérateurs sont polymorphes `<, <=` etc...
  - n-uplet de type type1 * type2 * ...

Résumé

- Fonctions

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)
  - Dans un `match` ou une déclaration de fonction sans paramètre

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)
  - Dans un `match` ou une déclaration de fonction sans paramètre
  - Les motifs peuvent contenir des identifiants et des constructeurs mais pas d'opérateurs

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)
  - Dans un `match` ou une déclaration de fonction sans paramètre
  - Les motifs peuvent contenir des identifiants et des constructeurs mais pas d'opérateurs
  - Les motifs sont traités les uns après les autres. Les expressions associées doivent toutes être du même type.

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)
  - Dans un `match` ou une déclaration de fonction sans paramètre
  - Les motifs peuvent contenir des identifiants et des constructeurs mais pas d'opérateurs
  - Les motifs sont traités les uns après les autres. Les expressions associées doivent toutes être du même type.
  - Le motif `_` "match" tout (peut donc être utilisé pour le cas par défaut)

Résumé

- Fonctions
  - À un paramètre `let f x = ...` , fonction anonyme `function x ->`
  - À plusieurs paramètres `let f (x, y, z...) = ...` version décurryfiée, `let f x y z ... =` version curryfiée, fonction anonyme `fun x y z ...->`
  - Récursivité : la fonction peut s'appeler elle-mêmen utilisation du mot clé `rec` : `let rec f ...`
- Filtrage (pattern-matching = mise en correspondance de motif)
  - Dans un `match` ou une déclaration de fonction sans paramètre
  - Les motifs peuvent contenir des identifiants et des constructeurs mais pas d'opérateurs
  - Les motifs sont traités les uns après les autres. Les expressions associées doivent toutes être du même type.
  - Le motif `_` "match" tout (peut donc être utilisé pour le cas par défaut)
  - Motifs peuvent être **gardés** pour plus d'expressivité des contraintes.

Résumé

- Polymorphisme : exemple `fst` et `snd` .

Résumé

- Polymorphisme : exemple `fst` et `snd` .
- Gestion des erreurs : `failwith "bla bla bla"`