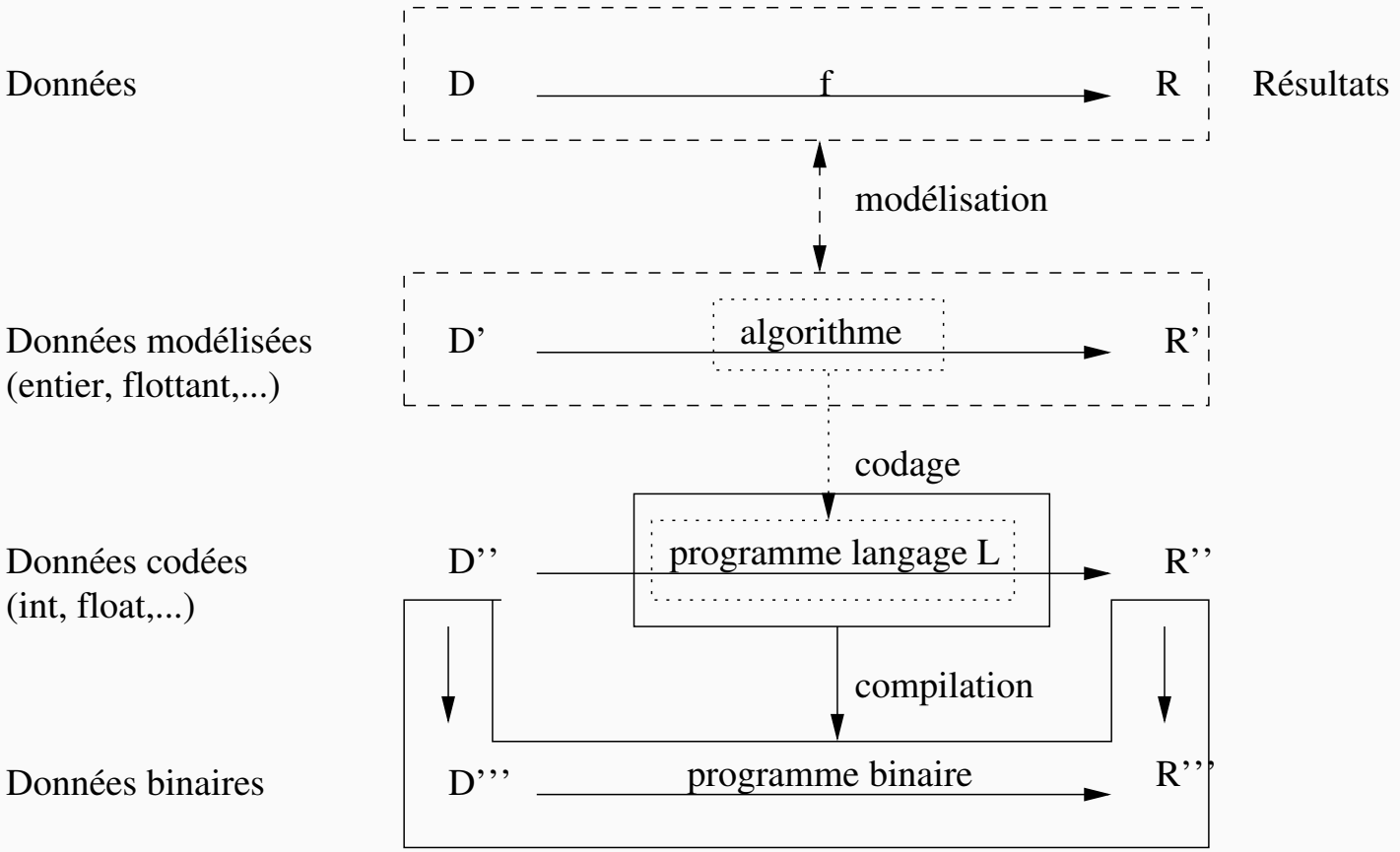
Cours 1 : Introduction à la programmation fonctionnelle et à OCaml

2019 - 2020

Introduction et historique ♪♪♪

Algorithmique et programmation fonctionnelle

- Description en termes de fonctions



Qu’est-ce que la programmation fonctionnelle ?

Programmation fonctionnelle vs impérative

- **programmation impérative**
  - modifications d’un état global (effets de bords) / impure
  - instructions
- **programmation fonctionnelle**
  - absence d’effet de bord / pure
  - tout est expression
- langage de programmation en général : **mixte**, choix entre une description en fonctionnel ou en impératif (simplicité)

Programmation fonctionnelle vs impérative

- **programmation impérative - impure**

```
j := 4;

g(i) {
  j := j+i;
  return j;
}

g(1) -> 5
g(1) -> 6
```

- **programmation fonctionnelle - pure**

```
f(i) = i+4;

f(1) = 5
f(1) = 5
```

Qu’est-ce que la programmation fonctionnelle ?

Conséquence de la pureté

- Indépendance au contexte de l’application d’une fonction
- Indépendance à l’ordre des applications dans les expressions constituées de fonctions pures et totales

Conséquence de l’indépendance au contexte et à l’ordre d’évaluation

- Formalisations facilitées de la notion de fonction (pas d’états)
- Typages plus complets et plus représentatifs du comportement des fonctions : toute sous-expression renvoie un résultat (typable).
- Parallélisation naturelle
- Lisibilité, maintenabilité améliorées
- Tests facilités : tests en boîte noire immédiat

Qu’est-ce que la programmation fonctionnelle ?

Langages de programmation

- Langages fonctionnels : Lisp, Scheme, SML (Standard ML), Caml (avec OCaml), Erlang, Haskell,. . .
- Langages multiparadigmes permettant une approche fonctionnelle : JavaScript, F#, Scala, Clojure, Java 8, Rust, Python. . .

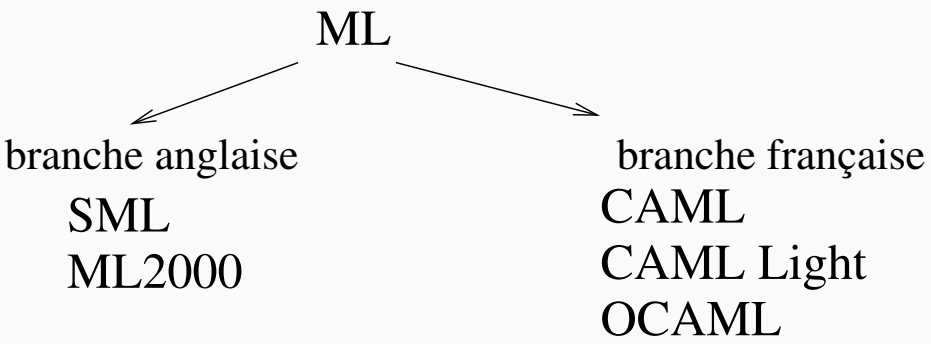
Évolution des langages

- introduction de traits fonctionnels dans des langages “mainstream”: typage, gestion mémoire, types algébriques, flux, abstractions, etc
- recommandation du style de programmation fonctionnelle

Le langage OCaml

Origines

- 1930 Alan Church : Théorie de la fonctionnalité ou λ-calcul
- 1964 John MacCarthy : Lisp Programmation fonctionnelle
- 1978 Robin Milner : ML (Méta Langage)



Le langage OCaml

OCaml est :

- un langage fonctionnel typé (avec des aspects impératifs et objets)
- disponible gratuitement Unix, Windows, PC, Mac,...

Type, typage

Caractéristiques du système de types d'OCaml ♪♪♪

Contrôle vs Inférence

- contrôle de types (C, JAVA, ADA, OCaml) : types explicités par le développeur et "vérifiés" par le compilateur / l'interprète.
- inférence de types (OCaml) : types calculés (et donc vérifiés).

Statique vs Dynamique

- typage statique (C, JAVA, ADA, OCaml) : les types sont déterminés et contrôlés statiquement, i.e. avant toute exécution.
- typage dynamique (JAVA, Python) : les types sont déterminés et contrôlés pendant l'exécution.

Fort vs faible

- typage fort (ADA, OCaml) : prémunit contre nombre d'erreurs.

  Un programme accepté fournit un résultat conforme à son type ou boucle mais n'adopte pas un comportement erratique nuisible.
- typage "faible" (C) : garantie peu claire.

  Vérification que les données manipulées ont la bonne taille en mémoire.

Éléments de base OCaml

♪♪♪

Tout est expression

- En OCaml, tout est **expression** (**entité qui a une valeur**).
  Une expression a un type et/ou une définition, suivant en cela la tradition mathématique.
- Il n'y a pas d'état, de changement d'état, de transformation.
- Il y a néanmoins des "variables" (mais dont la valeur ne change pas !)
  - elles apparaissent en tant que définitions globales ou locales, ou comme paramètres de fonctions.
  - Les variables, ou plutôt les identificateurs utilisés sont obligatoirement **déclarés** et **initialisés** (notion d'**environnement**)

Conventions et contraintes de nommage

- Un identificateur commence nécessairement par une lettre minuscule ou "_", suivi indifféremment de lettres minuscules ou majuscules, de chiffres, de "′ " ou "_".
- OCaml est sensible aux lettres majuscules et minuscules.

Évaluation

Exécution d'un programme

- L'**exécution d'un programme** : consiste à déterminer la valeur de l'expression que l'on a définie (**évaluation**).
- Les expressions manipulées sont généralement complexes et peuvent se décomposer en sous-expressions.
- Le principe de l'évaluation d'une expression composée consiste à évaluer d'abord ses sous-expressions, puis à calculer le résultat global.

Résultat d'une évaluation

- Terminaison correcte, i.e. fournir un résultat conforme à son type ;
- Boucle indéfiniment ;
- Arrêt brutal et levée d'une erreur (par exemple à cause d'une division par 0).

Structures de contrôle et Structures de données

Structures de contrôle

- Une structure de contrôle est une expression composée permettant de choisir quelles sous-expressions sont évaluées et dans quel ordre.
- Exemple (programmation fonctionnelle) : définition, conditionnelle, appel de fonction, filtrage
- Exemple (programmation impérative) : séquence, boucle Tant Que, boucle For

Structures de données

- Une structure de données est une expression composée permettant d'agréger les valeurs des sous-expressions dans une même donnée.
- Exemple : paires, n-uplets, listes, arbres

Expressions constantes, types primitifs et opérations

Type	Quelques valeurs	Quelques opérations
`bool`	`true false`	&& ‖ not
`int`	−1 0 5	+ - * /
`float`	0.0 5.5652 −0.000000001	+. -. *. /.
`char`	`'a' 'b' '5' '$'`	
`string`	`"toto" "" "Bonjour, maître"`	∧
`unit`	`()`	

Les opérateurs de comparaison (= < <= > >= <>) sont définis sur tous les types.

Expressions constantes et types primitifs

♪♪♪

Démo

Structure de contrôle : Définitions

♪♪♪

Définition : association d'un **nom (identificateur)** à la valeur d'une expression.

Plusieurs types de définitions[1]

- Définitions globales
- Définitions simultanées
- Définitions locales (temporaires)
- Emboîtement des définitions (globalité-localité)

Comportement des définitions

- Une définition locale masque temporairement les définitions globales de même nom.
- Une définition locale ne modifie pas une définition globale.
- Une définition locale dans un bloc *let* interne masque les définitions de même nom dans les blocs plus externes.

[1] Les ;; sont nécessaires uniquement dans l'interpréteur pour demander d'évaluer l'expression.

Structure de contrôle : Définitions

♪♪♪

Définition : association d'un **nom (identificateur)** à la valeur d'une expression.

Plusieurs types de définitions[1]

- Définitions globales

```
# let x=4;;
val x : int = 4
```

```
# x-1;;
- : int = 3
```

- Définitions simultanées
- Définitions locales (temporaires)
- Emboîtement des définitions (globalité-localité)

Comportement des définitions

- Une définition locale masque temporairement les définitions globales de même nom.
- Une définition locale ne modifie pas une définition globale.
- Une définition locale dans un bloc *let* interne masque les définitions de même nom dans les blocs plus externes.

Structure de contrôle : Définitions

♪♪♪

Définition : association d'un **nom (identificateur)** à la valeur d'une expression.

Plusieurs types de définitions[1]

- Définitions globales
- Définitions simultanées

```
# let y=2 and z=x-1;;
val y : int = 2
val z : int = 3
```

```
# x*x+y*y;;
- : int = 20
```

- Définitions locales (temporaires)
- Emboîtement des définitions (globalité-localité)

Comportement des définitions

- Une définition locale masque temporairement les définitions globales de même nom.
- Une définition locale ne modifie pas une définition globale.
- Une définition locale dans un bloc *let* interne masque les définitions de même nom dans les blocs plus externes.

Structure de contrôle : Définitions

♪♪♪

Définition : association d'un **nom (identificateur)** à la valeur d'une expression.

Plusieurs types de définitions[1]

- Définitions globales
- Définitions simultanées
- Définitions locales (temporaires)

```
# let x=7;;
val x : int = 7
# let x=4 in  2*x+1;;
- : int = 9
```

```
# x;;
- : int = 7
```

- Emboîtement des définitions (globalité-localité)

Comportement des définitions

- Une définition locale masque temporairement les définitions globales de même nom.
- Une définition locale ne modifie pas une définition globale.
- Une définition locale dans un bloc *let* interne masque les définitions de

Structure de contrôle : Définitions

♪♪♪

Définition : association d'un **nom (identificateur)** à la valeur d'une expression.

Plusieurs types de définitions[1]

- Définitions globales
- Définitions simultanées
- Définitions locales (temporaires)
- Emboîtement des définitions (globalité-localité)

```
# let x=3 in
          let y=2*x in
                    x+y;;
- : int = 9
```

```
# let x=3 in
          let x=1 in x;;
- : int = 1
```

Comportement des définitions

- Une définition locale masque temporairement les définitions globales de même nom.
- Une définition locale ne modifie pas une définition globale.
- Une définition locale dans un bloc *let* interne masque les définitions de

Notion d'environnement

Définition

- L'interprétation des déclarations globales-locales, éventuellement emboîtées, nécessite la notion d'environnement.
- Un **environnement de calcul** est une liste ordonnée de couples (*identificateur*, *valeur*) appelées **liaisons**.
- Notations et exemples :
  - $E = (y, 7); (x, 3); (z, 4); (x, 1)$
  - $E' = (u, 1); E$, i.e. $E' = (u, 1); (y, 7); (x, 3); (z, 4); (x, 1)$
  - $\emptyset$ désignera l'environnement vide

Calcul de valeur

- La **valeur** d'un identificateur x dans un environnement E, notée $\mathit{valeur}(x, E)$, est la valeur liée à x dans la **liaison située la plus à gauche** dans E.

$$E = (y, 7); (x, 3); (z, 4); (x, 1)$$
$$\mathit{valeur}(y, E) = 7$$
$$\mathit{valeur}(x, E) = 3$$

- On peut définir la *valeur* d'un identificateur dans un environnement comme suit :

$$\begin{aligned}
\mathit{valeur}(x, \emptyset) &= \texttt{indéfini} \\
\mathit{valeur}(x, (y, v); E) &= v \ \texttt{si} \ x = y \\
\mathit{valeur}(x, (y, v); E) &= \mathit{valeur}(x, E) \ \texttt{si} \ x \neq y
\end{aligned}$$

Exemples de calcul d'environnement

```
# let a=3;;
val a : int = 3          (a,3);l

# let x=a+1 in           (x,4);(a,3);l
    let y=1 in           (y,1);(x,4);(a,3);l
        x+y;;
- : int = 5              (a,3);l

# let a=-3 and z=a in    (z,3);(a,-3);(a,3);l
    z;;
- : int = 3              (a,3);l
```

Structure de contrôle : la conditionnelle

- Dans les langages fonctionnels, la définition est la structure de contrôle principale. Il en existe néanmoins d'autres, notamment la conditionnelle.
- La construction **conditionnelle** permet de définir une expression par cas, dont la valeur dépend d'une condition logique (i.e. une expression dont le type est booléen).
- Pour un typage correct, les deux sous-expressions "then" et "else" doivent être de même type.

Structure de contrôle : la conditionnelle

Syntaxe par l'exemple

```
# let a=-3;;
val a : int = -3

# if a>=0 then a else -a;;
- : int = 3

# let b = if a>=0 then a else -a;;
b : int = 3
```

Evaluation

- L'expression ($a >= 0$) est la condition.
- **Si** sa valeur est "vraie" dans l'environnement courant
  - **alors** l'expression entière a pour valeur la valeur de a (branche "**then**"),
  - **sinon** celle de $-a$ (branche "**else**").
- La conditionnelle n'évalue ainsi qu'une seule des deux sous-expressions "then" et "else" (construction **paresseuse**)

Spécification de fonction

- La **spécification** d'une fonction est constituée de l'ensemble

  contrat + test unitaire.

- Toute fonction définie doit être et sera nécessairement accompagnée de sa spécification.

Pourquoi spécifier une fonction?

- Types seuls ne permettent pas de comprendre le fonctionnement et les limites d'utilisation des fonctions.
- L'utilisateur de fonctions n'est pas nécessairement la personne qui les a écrites ou bien l'utilisateur n'a peut-être même pas accès au code source de ces fonctions.
- Il n'a pas obligatoirement envie de comprendre parfaitement le code avant de pouvoir l'utiliser en toute sérénité (sans erreurs non prévues)

$\Rightarrow$ **contrat** qui spécifie la **sémantique** de la fonction.

Quand spécifier ?

Cette spécification est à écrire avant d'écrire la fonction.[2]

[2] "avant" dans le sens temporel et non spatial

Contrat de fonction - contenu

Le contrat comprend :

- le nom **significatif** et son type.
- le rôle de la fonction, expliquée synthétiquement.
- le nom **significatif**, le type et le rôle des paramètres, **le cas échéant,** le domaine de validité des paramètres, pour lequel la fonction est bien définie (renvoie un résultat, cas **nominal**), i.e. la **précondition(s)**.
- le type et la spécification du résultat attendu en fonction des paramètres dans le cas nominal, i.e. la **postcondition(s)**.
- la liste des erreurs éventuelles prévues, toujours dans le cas nominal.

Contrat de fonction - rôle

Ce contrat est établi entre :

- l'utilisateur de la fonction, qui s'engage à respecter la précondition lors des appels ;
- le développeur de la fonction, qui s'engage alors à respecter la postcondition ou à ne lever que les erreurs prévues.

Si le contrat est violé par l'une des deux parties, alors l'appel de la fonction peut se comporter absolument n'importe comment.

Tests de fonction

- Toutes les fonctions définies doivent être testées individuellement afin de vérifier, dans la mesure du possible, si elles respectent leur part du contrat.
- On parle de **test unitaire**.
- Ces tests complètent la description de la fonction faite dans le contrat et doivent être écrits **avant** d'écrire la fonction.

Tests unitaires

Un test unitaire

- est composé de différents **cas de tests** : couples (arguments, résultat attendu).
- est capable de tester les différents comportements possibles de la fonction, à travers différentes situations typiques et significatives.
- peut être **automatisé** en exécutant la fonction sur les arguments proposés et en comparant le résultat obtenu au résultat attendu.

Les cas de tests doivent contenir au moins les cas terminaux et quelques cas génériques pour les fonctions récursives.

Exemple de spécification

```
(*
   fact : int -> int
   calcule la factorielle
   Parametre n : int, le nombre dont on veut la factorielle
   Resultat : int, factorielle de n
   Precondition : n strictement positf
*)
let rec fact n =
  if n = 1
  then 1
  else n * (fact (n-1))


(* OCaml permet d'ecrire simplement des tests unitaires (moyennant l'activation d'un
   preprocesseur). On peut alors utiliser la notation suivante: *)

let%test _ = fact 1 = 1
let%test _ = fact 2 = 2
let%test _ = fact 5 = 120
```

♪♪♪

Des expressions comme les autres

- OCaml est un langage où les fonctions sont des éléments ordinaires, au même titre que les entiers par exemple.
- On pourra donc avoir des fonctions qui prennent des fonctions en paramètre, appelées “fonctionnelles” en mathématiques.
- Comme toute donnée, une fonction possède un type.
- La seule opération permise sur les fonctions est l’appel.

Syntaxe par l'exemple

- Fonction qui calcule la valeur absolue d'un entier

```
# fun x -> if x >= 0 then x else -x;;
val - : int -> int = <fun>
```

- On peut associer un identificateur à une fonction en utilisant une définition :

```
# let valeur_absolue = fun x -> if x >= 0 then x else -x;;
val valeur_absolue : int -> int = <fun>
```

- Cette notation est couramment abrégée comme suit :

```
# let valeur_absolue x =
    if x >= 0 then x else -x;;
val valeur_absolue : int -> int = <fun>
```

- *x* est appelé **paramètre formel** de la fonction.

Les fonctions à plusieurs paramètres - Typage

```
(* test de la divisibilite de y par x *)
#let divise x y = y mod x = 0;;
val divise : int -> int -> bool = <fun>
```

- Le type `int -> int -> bool` exprime la présence des deux paramètres `int` et du résultat `bool`.
- Il faut en fait lire ce type comme
  `int -> (int -> bool)`,
- i.e. pour OCaml, cette fonction prend un paramètre de type `int` et renvoie une fonction de type `int -> bool`.
- On obtient ainsi une nouvelle fonction qui attend un (second) paramètre.

Les fonctions à plusieurs paramètres - Application partielle

- En exploitant ce fait lors de l'appel, on peut réaliser une **application partielle** de fonction.
- Exemple avec la fonction `pair`

```
(* test de parite *)
#let pair = divise 2;;
val pair : int -> bool = <fun>
```

- Attention l'ordre des paramètres détermine quelle application partielle est possible directement.

Appel de function

- Il s'écrit simplement en juxtaposant plusieurs expressions.
- Celle de gauche doit être une fonction, les suivantes sont les **arguments réels**.
- Les arguments réels doivent posséder un type compatible avec les types attendus des paramètres formels.
- Aucune parenthèse n'est nécessaire, sauf si les arguments réels sont eux-mêmes des expressions composées.

Exemples d'appels de fonction

```
#let divise x y = y mod x = 0;;
val divise : int -> int -> bool = <fun>

# divise 34 6;;
- : bool = false

# divise 2 (-42);;
- : bool = true

# let a = 3 in
  divise (a-5) (a+39);;
- : bool = true
```

Structure de contrôle

- L'appel de fonction est considéré comme une structure de contrôle en OCaml, car les arguments réels sont toujours évalués avant que la fonction ne soit "exécutée" à son tour.
- On dit que OCaml est un langage **strict**.

Composition de fonctions

Il est également possible de composer des fonctions :

```
# let carre x = x * x;;
val carre : int -> int = <fun>

# let puissance_cinq x =
x * carre (carre x);;
val puissance_cinq : int -> int = <fun>
```

Appel de fonction et environnement

- Le corps de la fonction est évalué dans l'environnement courant, auquel a été ajouté une nouvelle liaison entre le paramètre formel et la valeur de l'argument réel.
- Évaluer l'appel `carre (a+1)` dans l'environnement courant, contenant les définitions de `carre` et de `a` :
  - associer localement la **valeur** du paramètre réel `a+1` (i.e. 4) au paramètre formel *x* (**appel par valeur**).
  - la valeur de l'appel est alors la valeur de l'expression `x*x`, calculée dans cet environnement.
- On a l'équivalence entre `carre (a+1)` et `let x = (a + 1) in x * x`.
- Ces deux expressions correspondent au même calcul et donnent la même valeur.

Fonction et liaison statique

- Une fonction peut faire référence à un identificateur global :

```
# let y = 2;;
val y : int = 2
# let f x = x + y;;
val f : int -> int = <fun>
# f 3;;
- : int = 5
# let y = 6;;
val y : int = 6
# f 3;;
- : int = 5
```

- La fonction conserve le même sens malgré l'introduction d'un nouvel `y`.
- Une telle construction s'appelle une **fermeture fonctionnelle**.

Ordre supérieur

- Les fonctions étant des expressions comme les autres, elles peuvent être en paramètre d'autres fonctions.
- ```
  # let compose f g x = f (g x);;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ```
- Nous avons déjà vu avec l'exemple du test de parité que les fonctions peuvent également renvoyer des fonctions.
  ```

Polymorphisme

- La fonction compose précédente est dite **polymorphe** (plusieurs types)
- Les opérateurs de comparaison sont également polymorphes :
  val = : 'a -> 'a -> bool = <**fun**>

Syntaxe par l'exemple

- En OCaml, on dispose nativement de la **structure de données** n-uplets, ce qui en termes de types correspond au produit cartésien.

- ```
  #1,true;;
  - : int * bool = (1, true)
  #(2,false);;
  - : int * bool = (2, false)
  #(2,(1,3));;
  - : int * (int * int) = (2, (1, 3))
  #1,2,3;;
  - : int * int * int = (1, 2, 3)
  ```
  ```

Structure de données

- Les N-uplets sont des structures de données **composées**, i.e. toute donnée de ce type peut se décomposer en morceaux.
- Ces morceaux peuvent être récupérés à l'aide de fonctions : les **accesseurs**.
- Accès aux composantes d'une paire : Les fonctions `fst` (first) et `snd` (second).

```
#fst (1,2);;
- : int = 1
#snd (7,"toto");;
- : string = "toto"
#snd (1,(2,true));;
- : int * bool = (2, true)
#fst 1,2 ;;
Toplevel input:
>fst 1,2 ;;
>    ^
This expression has type int but is used with type 'a * 'b.
#fst (1,2,3);;
Toplevel input:
>fst (1,2,3);;
>    ^^^^^^^
This expression has type int * int * int but is used with type int * int.
```

♪♪♪

Retour sur les fonctions à plusieurs paramètres - Curryfication

- Les fonctions peuvent comporter plusieurs paramètres, envisagés séparément, ou bien réunis dans un n-uplet.
- Une fonction avec des paramètres réunis en n-uplet est une fonction à un seul paramètre de type n-uplet.
- Il vaut mieux éviter de manipuler des n-uplets, sauf si la réunion effective de *n* arguments dans un *n*-uplet a un sens.
- Le passage de la version à un paramètre de type n-uplet à la version à plusieurs paramètres s'appelle la **curryfication** (du nom du logicien américain Haskell Curry).

Pour les n-uplets autres que les paires, il n'existe pas d'accesseurs, il faut donc utiliser un autre mécanisme : **filtrage**.

Filtrage simple - Syntaxe par l'exemple pour les paires

- Exemple

```
#let t=1, (1, 2);;
t : int * (int * int) = (1, (1, 2))
#let (x,y)=t;;
x : int = 1
y : int * int = (1, 2)
```

- Le filtrage crée des liaisons entre les variables du filtre `(x,y)` et les valeurs des morceaux de la paire représentée par l'identificateur `t`.
- On a l'équivalence avec `let x=(fst t) and y=(snd t)`.

Filtrage simple - Généralisation aux N-uplets

- La possibilité de filtrage n'est pas limitée aux paires :
- Exemple

```
#let (x,y,z)=(1,"toto",(5,6));;
x : int = 1
y : string = "toto"
z : int * int = (5, 6)
```

Filtrage simple - linéarité

- Le filtrage en OCaml correspond uniquement à des créations de liaisons. Doubler une variable dans un filtre n'a pas de sens :

```
let (x,x) = (a,b)
```

- Cela équivaudrait à essayer de définir en même temps `x` avec les valeurs potentiellement différentes `fst (a,b)` et `snd (a,b)`, i.e. :

```
let x = fst (a,b)
and x = snd (a,b)
```

- Cette situation est en fait interdite, nous avons affaire à un filtrage appelé **linéaire**.

Filtrage simple - filtrage partiel

- Un filtrage partiel est réalisé avec _ et ne crée pas de liaison (rien n'est ajouté dans l'environnement) :

```
#let (x,_,z)=(1,"toto",(5,6));;
x : int = 1
z : int * int = (5, 6)
#let _=(1,"toto");;
```

- Cela ne permet pas pour autant de filtrer n'importe quoi !

```
#let (x,_,z)=(1,"toto");;
Toplevel input:
>let (x,_,z)=(1,"toto");;
>            ^^^^^^^^^^
This expression has type int * string but is used with type 'a * 'b * 'c.
```

Filtrage par cas et échec - Syntaxe

- Le mécanisme de filtrage se généralise au raisonnement par cas.
- Cette généralisation est représentée par la structure de contrôle `match... with...` :

```
match expression with
  | F_1 -> resultat_1
  | F_2 -> resultat_2
   ...
  | F_n -> resultat_n
```

Filtrage par cas et échec - Sémantique

```
match expression with
  | F_1 -> resultat_1
  | F_2 -> resultat_2
   ...
  | F_n -> resultat_n
```

- Si la valeur de l'expression est filtrée par F_1, alors l'expression $resultat_1$ est évaluée,
- sinon si la valeur de l'expression est filtrée par F_2, alors l'expression $resultat_2$ est évaluée,
- $\vdots$
- sinon si la valeur de l'expression est filtrée par F_n, alors l'expression $resultat_n$ est évaluée,
- sinon c'est un **échec** de filtrage.

Filtrage par cas et échec - Les filtres

- Contrairement à la conditionnelle, on ne peut pas tester comme condition de filtrage l'égalité à la valeur d'une variable.
- Un filtre est composé de variables libres ou de constantes seulement.
- Les filtres doivent avoir des types "compatibles" : le type attendu de l'expression filtrée. De même les résultats doivent eux aussi tous être de même type.

Filtrage par cas et échec - Exemple

```
# let famille animal =
    match animal with
    | "poule"   -> "oiseau"
    | "chat"    -> "mammifere"
    | "chien"   -> "mammifere"
    | "daurade" -> "poisson"
    | _         -> "inconnu";;
famille : string -> string = <fun>

# let vecteur_pur v =
    match v with
    | (0.0, _  ) -> true
    | (_  , 0.0) -> true
    | _          -> false;;
vecteur_pur: float * float -> bool = <fun>

# let premier nuplet =
    match nuplet with
    | (f, _)       -> f
    | (f, _, _)    -> f
    | (f, _, _, _) -> f
    | f            -> f;;

    | (f, _, _)    -> f
      ^^^^^^^^^
Error: This pattern matches values of type 'a * 'b * 'c
       but a pattern was expected which matches values of type 'd * 'e
```

Filtrage par cas et échec - Échec du filtrage

- L'échec de filtrage est considéré comme une erreur à l'exécution, celle-ci s'interrompt brutalement, et l'erreur est signalée.
- Afin d'éviter ce cas d'erreur, il faut impérativement définir un filtrage **total**, au besoin en ajoutant un cas terminal _ qui filtre toute donnée non filtrée par les cas précédents.

Importance du filtrage

Nous n'avons vu ici qu'une introduction au filtrage. Il s'agit en fait d'une construction très puissante qui sera grandement généralisée par la suite.

Erreurs et exceptions

- L'exécution d'un programme peut lever différentes erreurs, appelées **exceptions** :
  - par le programme, en cas par exemple de division par zéro ou d'échec de filtrage.
  - par l'utilisateur, si l'exécution ne peut se poursuivre et donner un résultat cohérent/significatif.
- En règle générale, il vaut mieux lever une exception que renvoyer une valeur "par défaut" non significative.

Failwith

- L'utilisateur peut interrompre l'exécution en appelant la fonction spéciale `failwith : string -> 'a`
- Lorsqu'elle est appelée (avec pour argument un message d'erreur), elle affiche ce message et arrête l'exécution en cours.

```
# let famille animal =
    match animal with
    | "poule"   -> "oiseau"
    | "chat"    -> "mammifere"
    | "chien"   -> "mammifere"
    | "daurade" -> "poisson"
    | _         -> failwith "inconnu";;
famille : string -> string = <fun>
```

Comme dans beaucoup de langage de programmation, il est possible de définir des exceptions, de les lever et des récupérer / traiter.

Exception - Définition - Syntaxe par l'exemple

- ```
  # exception AnimalInconnu;;
  exception AnimalInconnu
  # exception DateInvalide of (int*int*int);;
  exception DateInvalide of (int*int*int)
  ```
  ```

Lever une exception - Syntaxe par l'exemple

- Les exceptions sont levées par la fonction `raise`.

```
# let famille animal =
    match animal with
    | "poule"   -> "oiseau"
    | "chat"    -> "mammifere"
    | "chien"   -> "mammifere"
    | "daurade" -> "poisson"
    | _         -> raise AnimalInconnu;;
val famille : string -> string = <fun>
# famille "cheval";;
Exception: AnimalInconnu.

# let getMois date =
  match date with
   | (_,1,_) -> "Janvier"
   | (_,2,_) -> "Fevrier"
   | ...
   |(_,12,_) -> "Decembre"
   | _ -> raise (DateInvalide date);;
val getMois : int * int * int -> string = <fun>
# getMois (4,13,2018);;
Exception: DateInvalide (4, 13, 2018).
```

Récupérer une exception - Syntaxe par l'exemple

- Les exceptions sont récupérées par la structure `try ... with`.

```
# let printer animal date =
  try
     "J'ai␣adopte␣mon␣"^animal^"␣(famille␣des␣"^famille animal^")␣en␣"^getMois date
  with
   | AnimalInconnu
       -> "J'ai␣adopte␣un␣animal␣inconnu"
   | DateInvalide (_,m,_)
       -> "Je␣vis␣dans␣un␣autre␣espace␣temps␣ou␣il␣existe␣un␣mois␣"^(string_of_int m);;
val printer : string -> int * int * int -> string = <fun>

# printer "chat" (05,12,2015);;
- : string = "J'ai␣adopte␣mon␣chat␣(famille␣des␣mammifere)␣en␣Decembre"

# printer "cheval" (05,12,2015);;
- : string = "J'ai␣adopte␣un␣animal␣inconnu"

# printer "chat" (02,14,2015);;
- : string = "Je␣vis␣dans␣un␣autre␣espace␣temps␣ou␣il␣existe␣un␣mois␣14"
```

Les modules

Objectif

- Décrire séparément la spécification et l'implantation

Une version simplifiée des modules

- Spécification : fichier `.mli`
  - Déclaration des types
  - Nom des symboles visibles définis dans l'implantation
  - Types de ces symboles
  - Contrats de ces symboles
- Implantation : fichier (même nom) `.ml`
  - Définition des types
  - Implantations des fonctions / symboles définis dans la spécification
- Similaire à la notion d'interface et de réalisation en Java

Exemple - Spécification - date.mli

```
type date

(* Fonction qui renvoie le nom du mois d'une date *)
val getMois : date -> string
```

Exemple - Implantation - date.ml

```
type date = (int * int * int)

let getMois date =
  match date with
  | (_,1,_) -> "Janvier"
  | (_,2,_) -> "Fevrier"
  | ...
  |(_,12,_) -> "Decembre"
  | _ -> raise (DateInvalide date)
```