Cours 1 : Introduction à la programmation fonctionnelle et à OCaml

1 Introduction et historique

1.1 Qu'est-ce que l'informatique ?

Informatique : Science du traitement automatique et rationnel de l'information [...]. (Larousse)

L'informatique manipule et traite des données.

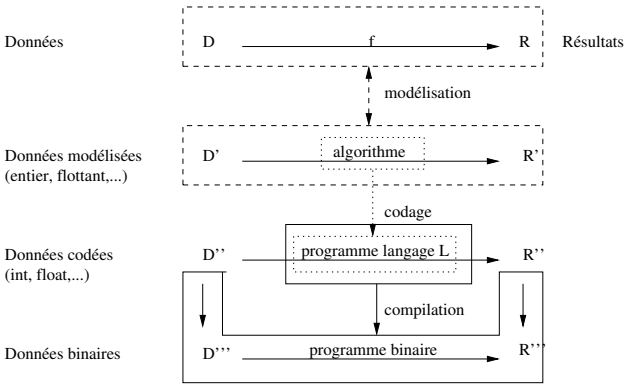
Description de ce traitement : fonction qui définit la valeur des sorties par rapport aux valeurs des entrées.

Problèmes concernant la réalisation de ce traitement à l'aide d'un ordinateur :
- représentation des entrées et des sorties dans un ordinateur (qu'est-ce qu'une information/donnée, comment la représente-t-on ?) ;
- représentation du traitement à proprement parler.

Domaines qui s'intéressent à résoudre ces problèmes :
- l'algorithmique : où l'on manipule des concepts communs issus à la fois de l'expérience humaine et des limites physiques des ordinateurs : variables, procédures, paramètres, valeurs, structures de contrôle, ...
- la programmation : où l'on choisit un sous-ensemble de concepts clés associés à une syntaxe/un langage spécifique, permettant la réalisation des traitements de données. Un programme est la traduction d'un algorithme dans un langage de programmation donné.

1.2 Qu'est-ce que la programmation fonctionnelle ?

Dans ce cadre, l'algorithmique et la programmation fonctionnelle font le choix de rester au plus près de la description en termes de fonctions mathématiques de ce qu'est un traitement de données. On peut alors établir la correspondance suivante :

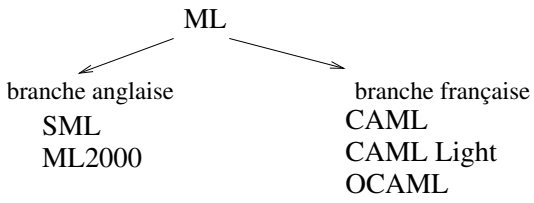


La programmation fonctionnelle est souvent opposée à la programmation impérative, où l'on décrit les traitements de données à l'aide d'instructions, qui représentent des modifications d'un état global, encore appelées effets de bord. Ainsi, la sortie d'un traitement sera exprimée comme une modification de son entrée, produite par le programme impératif.

Au contraire, le qualificatif "fonctionnel" évoque l'absence d'effets de bord. La plupart des langages de programmation sont néanmoins mixtes et intègrent des concepts issus de ces deux univers, certains traitements étant plus simplement décrits en fonctionnel qu'en impératif ou réciproquement.

Le langage OCaml OCaml est le langage que nous allons utiliser dans le cadre de ce cours. Voici ses origines :

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



OCaml est :

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

2 Interprète et compilateur

Tout langage de programmation, qui est un mode de description de traitement de données, doit être outillé, i.e. fournir un moyen de passer d'une description textuelle (syntaxique) du traitement à une description opérationnelle compréhensible par un ordinateur (en langage machine donc).

Un **interprète** est un outil interactif ayant pour tâche d'analyser, de traduire et d'exécuter les programmes écrits dans un langage informatique. Un **compilateur** est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible). Pour qu'il puisse être exploité par la machine, le compilateur traduit le code source, écrit dans un langage de haut niveau d'abstraction, facilement compréhensible par l'humain, vers un langage de plus bas niveau, un langage d'assemblage ou langage machine. Un interpréteur se distingue d'un compilateur par le fait que, pour exécuter un programme, les opérations d'analyse et de traductions sont réalisées à chaque exécution du programme (par un interpréteur) plutôt qu'une fois pour toutes (par un compilateur).

Nous allons utiliser dans un premier temps l'interprète de OCaml, qui à chaque entrée effectue les opérations suivantes :

1. lire le programme (vérification syntaxique)
2. typer ce programme (vérification sémantique)
3. exécuter ce programme
4. afficher le résultat obtenu et retourner au 1)

En fin de module nous utiliserons le **compilateur**, notamment pour le projet.

3 Type, typage

3.1 Qu'est-ce qu'un type ?

Toute donnée manipulée en OCaml possède un type (unique) qui représente l'**ensemble des valeurs** que pourra prendre cette donnée et spécifie également l'usage le plus général que l'on peut faire de cette donnée, i.e. par exemple les **opérations permises** la concernant. Les fonctions possèdent également un type qui spécifie le type des paramètres et du résultat.

3.2 Pourquoi typer ?

La programmation étant une activité où les erreurs peuvent être nombreuses et difficiles à déterminer (est-ce que le programme écrit réalise bien le traitement souhaité ?), les types servent à mieux spécifier/contrôler ce qui doit se produire lorsqu'on exécute un programme et notamment empêcher les comportements incohérents.

3.3 Caractéristiques d'un système de types

Contrôle vs inférence

— contrôle de types (C, JAVA, ADA, OCaml) : types explicités par le développeur et "vérifiés" automatiquement par un outil (intégré au compilateur ou à l'interprète).
— inférence de types (OCaml) : types calculés (et donc vérifiés) par un outil automatique.

En OCaml, les types peuvent être explicités par le développeur, ou inférés automatiquement par l'environnement de développement (interprète, etc).

Statique vs dynamique

— typage statique (C, JAVA, ADA,OCaml) : les types sont déterminés et contrôlés statiquement, i.e. avant toute exécution, justement pour vérifier que celle-ci se déroulera "correctement".
— 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. Grossièrement, si un programme est accepté, alors son exécution peut fournir un résultat conforme à son type ou bien encore boucler (i.e. ne jamais répondre) mais ne va pas adopter un comportement erratique nuisible par exemple au bon fonctionnement global de l'ordinateur.
— typage "faible" (C) : garantie peu claire. En C, consiste principalement à vérifier que les données manipulées ont la bonne taille (en nombre d'octets occupés en mémoire). Des changements de type peuvent être effectués implicitement.

4 Eléments de base OCaml

En OCaml, tout est **expression** (**entité qui a une valeur**) ayant un type ou définition, suivant en cela la tradition mathématique. Il n'y a pas d'état, de changement d'état, de transformation (bien qu'on utilise parfois ces termes par commodité de langage). Il y a néanmoins des "variables" (mais dont la valeur ne change pas !) qui 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).

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

Les expressions manipulées sont généralement complexes et peuvent se décomposer en sous-expressions (comme dans $a + b$ où a et b sont des sous-expressions).

Exécution d'un programme : consiste, dans un langage fonctionnel, à déterminer la valeur de l'expression que l'on a définie. On parle également d'**évaluation**. Le principe de l'évaluation d'une expression composée consiste à évaluer d'abord ses sous-expressions, puis à calculer le résultat global. Toute évaluation peut :

— se terminer correctement, i.e. fournir un résultat conforme à son type ;
— boucler indéfiniment ;
— s'arrêter brutalement et lever une erreur (par exemple à cause d'une division par 0).

Structures de contrôle : Une structure de contrôle est une expression composée permettant de choisir quelles sous-expressions sont évaluées et dans quel ordre.

Structures de données : Une structure de données est une expression composée permettant d'agréger les valeurs des sous-expressions dans une même donnée. C'est le cas des paires, des listes, des arbres, etc.

4.1 Expressions constantes et types primitifs

```
cobalt:~>ocaml
Objective Caml version 3.11.2
# 3+4;;              (* calcul d'une expression *)
- : int = 7

# exit 0;;           (* fonction de sortie *)
cobalt:~>


------------------------------------------------------
>               invite d'Unix
#               invite d'Ocaml
;;              fin de l'expression (nécessaire uniquement dans l'interpréteur)
int             type
7               valeur

# (3*5)/2;;
- : int = 7         (division dans N)

# 1.2+3;;
  ^^^
Error: This expression has type float but an expression was expected of type int

# 1.3+.3.1;;
- : float = 4.4

# 1.3+3.1;;
  ^^^
Error: This expression has type float but an expression was expected of type int
```

OCaml est **fortement typé** et calcule le type du résultat avant d'évaluer l'expression.

— "+", "-", "*", "/" sont les opérateurs du type *int*
— "+.", "-.", "*.", "/." sont les opérateurs du type $float$

Quelques exemples de valeurs et leur type.

type	**valeurs**
`bool`	`true false`
`int`	-1 0 5
`float`	0.0 5.5652 -0.000000001
`char`	`'a' 'b' '5' '$'`
`string`	`"toto" "" "Bonjour, maître"`
`unit`	`()`

4.2 Structure de contrôle : définitions globales et locales

On peut associer un `nom (identificateur)` à la valeur d'une expression.

— Définitions globales :

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

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

— 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) :

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

— 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
```

En conclusion :

— 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.

4.3 Notion d'environnement

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

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

$$E = (y, 7); (x, 3); (z, 4); (x, 1)$$
$$valeur(y, E) = 7$$
$$valeur(x, E) = 3$$

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

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

4.4 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.

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

# let absa = if a>=0 then a else -a;; (* une expression qui calcule la valeur absolue de a *)
absa : int = 3
```

L'expression ($a >= 0$) est la condition. **Si** sa valeur est "vraie" dans l'environnement courant (ce qui n'est pas le cas ici), **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", il s'agit d'une construction **paresseuse** (par opposition à un opérateur **strict**, qui évalue toutes ses sous-expressions, comme l'addition par exemple).

Les principaux opérateurs logiques prédéfinis sont les opérateurs relationnels, ainsi que les opérateurs booléens.

```
# let a=3;;
val a : int = 3
# let b=6;;
val b : int = 6

# a>b;;
- : bool = false

# b-1=a+2;;
- : bool = true

# (a>b) && (b-1=a+2);;
- : bool = false
```

- "<", ">", "=", ">=", "<=", "<>" sont les opérateurs relationnels, qui comparent deux données (expressions) de même type et renvoie un élément de type *bool*
- "&&", "||", "not" sont les opérateurs du type *bool*

4.5 Les fonctions

4.5.1 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.

D'une part, très rapidement, l'usage des types seuls ne permet pas de comprendre le fonctionnement et les limites d'utilisation des fonctions. Par exemple, toutes les opérations sur les fractions ont le même type, alors que leurs comportements sont très différents et peuvent occasionner des erreurs (par exemple ? Division par zéro, débordement arithmétique). D'autre part, l'utilisateur de fonctions (celui qui veut les appeler avec des arguments réels) 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. Pour que l'utilisateur ne soit pas obligé de comprendre parfaitement le code avant de pouvoir l'utiliser en toute sérénité (sans provoquer d'erreurs non prévues à l'exécution), il est alors obligatoire de fournir avec toute fonction un **contrat** qui spécifie la **sémantique** de la fonction :

Ce contrat, à écrire avant d'écrire la fonction, 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**.
- le type et la spécification du résultat attendu en fonction des paramètres dans le cas nominal, i.e. la **postcondition**.
- la liste des erreurs éventuelles prévues, toujours dans le cas nominal.

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.

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**. Ce test complète la description de la fonction faite dans le contrat et doivent être écrits avant d'écrire la fonction.

Un tel test est composé de différents **cas de tests**, i.e. de couples (arguments, résultat attendu) qui servent à tester les différents comportements possibles de la fonction, à travers différentes situations typiques et significatives. L'avantage d'un cas de test par rapport au contrat réside dans le fait que sa validité peut être **automatisée** 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.

4.5.2 Définition de fonction

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.

```
# fun x -> x*x;; (* la fonction qui élève un entier au carré *)
- : int -> int = <fun>

# fun x -> if x >= 0 then x else -x;; (* la fameuse valeur absolue *)
- : int -> int = <fun>
```

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

On peut associer un identificateur à une fonction :

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

(* ou bien *)
```

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

Il existe pour les fonctions deux écritures, la seconde est plus courante. On peut également donner explicitement le type du paramètre pour imposer l'usage de la fonction.

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

# let valeur_absolue_float (x : float) =
   if x >= 0 then x else -x;;
# let valeur_absolue_float (x : float) = if x >= 0 then x else -x;;
                                                     ^
Error: This expression has type int but an expression was expected of type float

# let valeur_absolue_float (x : float) =
   if x >= 0.0 then x else -.x;;
val valeur_absolue_float : float -> float = <fun>
```

4.5.3 Les fonctions à plusieurs paramètres

Les fonctions peuvent comporter plusieurs paramètres.

Exemple

```
(* test de la divisibilité 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.

Attention `->` est associative à droite mais pas à gauche : `int -> int -> bool` est équivalent à `int -> (int -> bool)` mais pas à `(int -> int) -> bool`. Ce dernier type correspond à une fonction qui prend en paramètre une fonction (qui prend en paramètre un entier et renvoie un entier) et renvoie un booléen.

La syntaxe "classique" des fonctions met mieux en évidence le fait qu'une fonction à n paramètres est une imbrication de n fonctions à paramètre unique :

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

(* c'est-à-dire *)
# let divise = fun x -> (fun y -> y mod x = 0);;

(* ou encore *)
# let divise = fun x y -> y mod x = 0;;
```

En exploitant ce fait lors de l'appel, on peut réaliser une **application partielle** de fonction :

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

(* c'est-à-dire *)
# let pair = fun x -> divise 2 x;;

(* ou par application partielle *)
# let pair = divise 2;;

# pair 3;;
- : bool = false

# pair 4;;
- : bool = true
```

Cette dernière écriture de la fonction `pair` par application partielle de la fonction `divise` (on ne lui fournit qu'un seul argument réel) peut se paraphraser : "être pair signifie être divisible par 2".

4.5.4 Structure de contrôle : Appel de fonction

Il s'écrit simplement en juxtaposant plusieurs expressions. Celle de gauche doit être une fonction, les suivantes les arguments réels. Les arguments réels doivent posséder un type compatible avec les types attendus des paramètres formels de la fonction. Aucune parenthèse n'est nécessaire, sauf si les arguments réels sont eux même des expressions composées. En cas de doute, mieux vaut mettre des parenthèses.

```
# (* test de la divisibilité de y par x *)
#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
```

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**.

4.5.5 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>
```

4.5.6 Appel de fonction et environnement

L'appel de fonction est interprété comme suit : on évalue le corps de la fonction (sa définition) dans l'environnement courant, auquel on a ajouté une nouvelle liaison entre le paramètre formel et la valeur de l'argument réel.

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

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

# carre (a + 1);;
- : int = 16

# let x = (a + 1) in x * x;;
- : int = 16
```

Dans cet exemple, évaluer l'appel `carre (a+1)` dans l'environnement courant, contenant les définitions de `carre` et de `a`, revient donc à :

1. associer localement la **valeur** du paramètre réel `a+1` (i.e. 4) au paramètre formel x (**appel par valeur**).
2. la valeur de l'appel est alors la valeur de l'expression `x*x`, calculée dans cet environnement.

Ainsi, on a l'équivalence entre `carre (a+1)` et `let x = (a + 1) in x * x` dans cet exemple. Ces deux expressions correspondent au même calcul et donnent la même valeur.

4.5.7 Fonction et liaison statique

Une fonction peut faire référence à un identificateur global (ici le premier `y`) :

```
# 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
```

On constate que la fonction `f` conserve toujours le même sens (il s'agit d'ajouter la valeur du premier `y`, i.e. 2 au paramètre `x`), quel que soit le contexte et l'introduction d'un nouvel `y`. On dit que les liaisons en OCaml sont **statiques**, i.e. déterminées une fois pour toutes à la définition de la fonction. Tout se passe comme si `f` "transportait" la valeur de `y`, i.e. comme si on avait en fait défini :

```
let f =
 let y = 2
 in fun x -> x + y
```

4.5.8 Ordre supérieur

Les fonctions étant des expressions comme les autres, il est possible d'écrire des fonctions qui prennent en paramètre des fonctions.

Exemple :

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

Nous avons déjà vu avec le curryfication que les fonctions peuvent renvoyer des fonctions.

4.5.9 Rappel : Programmation offensive vs Programmation défensive

Les cas non valides doivent toujours être prévus et apparaître dans le contrat d'une fonction. Néanmoins, l'établissement d'un contrat laisse beaucoup de liberté sur le traitement des situations non valides. On distingue deux approches.

Programmation offensive : le principe consiste à interdire au maximum les situations non valides, à l'aide d'une précondition plus restrictive pour l'utilisateur. Ce style laisse la responsabilité à l'appelant de vérifier les préconditions en amont de l'appel à la fonction.

Programmation défensive : le principe est de ne pas faire confiance à l'utilisateur de la fonction et ainsi de tester tous les cas de validité soi-même à l'intérieur de la fonction. Les cas non valides sont détectés et correspondent à une erreur prévue et documentée. Ce style laisse la responsabilité à l'appelé de lever des exceptions.

Remarque : la récupération des erreurs/exceptions ne sera pas abordée dans ce cours, étant donnée que les erreurs rattrapables intelligemment sont pour la plupart liées aux entrées-sorties, qui seront abordées en PIM (par exemple, une division par zéro ou un débordement ne sont pas rattrapables, alors qu'une lecture ratée au clavier pourra toujours être recommencée).

En général, on préfèrera la programmation offensive, sauf dans les cas où la précondition résultante est difficile à décrire ou trop complexe.

4.6 Structure de donnée : N-uplets

4.6.1 Construction de n-uplets

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

— par 2 (en paire) :
```
#1,true;;
- : int * bool = (1, true)
#(2,false);;
- : int * bool = (2, false)
#(2,(1,3));;
- : int * (int * int) = (2, ( 1, 3))
```
— ... ou plus :
```
#1,2,3;;
- : int * int * int = (1, 2, 3)
```

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**.

4.6.2 Accès aux composantes d'une paire

Les fonctions `fst` (first) et `snd` (second) permettent d'accéder respectivement à la première et à la seconde composante d'une paire.

```
#fst (1,2);;
- : int = 1
#snd (7,"toto");;
- : string = "toto"
#snd (1,(2,true));;
- : int * bool = (2, true)
#fst 1,2 ;; (* Ici les parenthèses sont obligatoires *)
Toplevel input:
>fst 1,2 ;; (* Ici les parenthèses sont obligatoires *)
>    ^
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.
```

4.6.3 Retour sur les fonctions à plusieurs paramètres

Les fonctions peuvent comporter plusieurs paramètres, envisagés séparément. Les paramètres peuvent aussi être réunis dans un n-uplet. Dans ce dernier cas, on dit que la fonction ne possède qu'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. Par exemple, un point 2D est une paire (abscisse, ordonnée) et les opérations sur des points auront naturellement intérêt à prendre de telles paires en paramètres/résultats.

Par contre, le pgcd de deux nombres est clairement une fonction à deux paramètres indépendants x et y, construire la paire (x, y) n'aurait aucun sens (que représente cette paire ? Rien). En conclusion, il vaut mieux définir :

```
let pgcd x y = ...
pgcd : int -> int -> int = <fun>
```

plutôt que la version que l'on trouve classiquement dans des ouvrages mathématiques :

```
let pgcd (x, y) = ...
pgcd : (int * int) -> int = <fun>
```

Le passage de la version classique à la "bonne" version s'appelle la **curryfication** (du nom du logicien américain Haskell Curry).

4.7 Filtrage et structure de contrôle match-with

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

4.7.1 Filtrage simple

```
#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)`.

Cette possibilité de filtrage n'est pas limitée aux paires :

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

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)
```

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, il s'agit d'un filtrage appelé **linéaire**.

Filtrage partiel Un tel filtrage 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, l'expression suivante n'est pas acceptée durant la phase de typage par OCaml. C'est une expression qui n'a pas de sens pour le langage :

```
#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.
```

Une paire n'est pas un triplet, et plus encore, les trois expressions suivantes n'ont pas le même type, on obtiendra donc une erreur de filtrage :

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

#((1,2),3);;
-:(int * int) * int = ((1, 2), 3)

#(1,(2,3));;
-:int * (int * int) = (1, (2, 3))

#let (x,y,z) = (1,(2,3));;
>let (x,y,z) = (1,(2,3));;
>              ^^^^^^^^^
This expression has type int * (int * int) but is used with type 'a * 'b * 'c.
```

4.7.2 Filtrage par cas et échec ("match... with ... ")

Le mécanisme de filtrage se généralise au raisonnement par cas, mais contrairement à la conditionnelle, on ne peut pas tester comme condition de filtrage l'égalité à une constante. Ainsi, un filtre est composé de variables ou de constantes seulement.

Cette généralisation est représentée par la structure de contrôle `match... with...` :

```
match expression with
  | F_1 -> resultat_1 (* la barre avant le F_1 est facultative, mais plus jolie *)
  | F_2 -> resultat_2
   ...
  | F_n -> resultat_n
```

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

Quelques exemples :

```
# let famille animal =
   match animal with
   | "poule"   -> "oiseau"
   | "chat"    -> "mammifère"
   | "chien"   -> "mammifère"
   | "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
```

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.

4.8 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.

4.8.1 Failwith

Le second cas peut être réalisé par la fonction spéciale `failwith : string -> 'a`, qui lorsqu'elle est appelée (avec pour argument un message d'erreur), affiche ce message et arrête l'exécution en cours.

Par exemple, la fonction `famille` définie précédemment devrait être remplaçée par :

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

qui évite que les animaux non reconnus soient tous silencieusement et abusivement placés dans la famille `"inconnu"`, qui elle-même n'existe pas. En règle générale, il vaut mieux lever une exception que renvoyer une valeur “par défaut” non significative.

4.8.2 Définir une exception

Le `failwith` entraîne l'arrêt du programme. Il est également possible de lever une exception qui pourra être récupérée.

Les exceptions sont déclarées par le mot clef `exception`. Attention en Ocaml le nom d'une exception doit commencer par une majuscule. Il est possible de définir un paramètre à une exception.

```
# exception AnimalInconnu;;
```

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

4.8.3 Lever une exception

Les exceptions sont levées par la fonction `raise`.

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

```
# let getMois date =
  match date with
  | (_,1,_) -> "Janvier"
  | (_,2,_) -> "Février"
  | ...
  |(_,12,_) -> "Décembre"
  | _ -> raise (DateInvalide date);;
# getMois (4,13,2018);;
Exception: DateInvalide (4, 13, 2018).
```

4.8.4 Récupérer une exception

Les exceptions peuvent être récupérées à l'aide d'un `try...with`. Dans le bloc `with` il est possible de filtrer les différentes exceptions qui ont pu être levées.

```
# let printer animal date =
  try
     "J'ai adopte mon "^animal^" (famille des "^famille animal^") en "^getMois date
  with
   | AnimalInconnu
       -> "J'ai adopté un animal inconnu"
   | DateInvalide (_,m,_)
       -> "Je vis dans un autre espace temps où il existe un mois "^(string_of_int m);;
val printer : string -> int * int * int -> string = <fun>
# printer "chat" (05,12,2015);;
- : string = "J'ai adopté mon chat (famille des mammifère) en Décembre"
# 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"
```

5 Les modules

— Un programme est rarement composé d'un fichier unique contenant toutes les définitions nécessaires, mais est le plus souvent décomposé en différents fichiers ou plus généralement en **unités de compilation**.
— Les définitions contenues dans une unité vont naturellement dépendre d'autres définitions dans d'autres unités.
— Ces unités n'étant pas écrites et maîtrisées par une seule et même personne, il est obligatoire de disposer d'une spécification pour chaque unité, qui résume le contenu et le rôle.
— Non seulement pour l'utilisateur, mais encore pour le support du langage lui-même, qui doit autoriser le développement de chaque unité indépendamment, que le reste ait été écrit ou non.

Il s'agit donc de décrire séparement la spécification et l'implantation. On peut se souvenir des ensembles (spécification) implantés par des listes (implantation).

Quels sont les avantages de séparer implantation et spécification :
— changer d'implantation de façon transparente pour un programmeur utilisant ce type,
— compilation séparée : voir TP,
— cacher les fonctions auxiliaires,
— cacher/vendre l'implantation.

Un couple (spécification, implantation) constitue un **module**, dans de nombreux langages de programmation. Deux modules différents peuvent donc respecter la même spécification tout en proposant une implantation différente. La spécification d'un module constitue une forme de contrat entre :
— le développeur, qui s'engage à développer ce qui figure dans la spécification.
— l'utilisateur, qui s'engage à n'utiliser que ce qui est déclaré dans la spécification.

Donc, plus la spécification contient d'informations, plus le contrat est précis, moins les chances de se tromper lorsqu'on effectue un développement collaboratif (majorité des cas) sont élevées.

En général, une spécification dans un langage de programmation contient :
— des définitions ou déclarations de types.
— le nom des symboles visibles définis dans l'implantation.
— le type de ces symboles, qui permet une connaissance grossière de l'utilisation possible des symboles. Cette information suffit en général aux compilateurs (ou interprètes) pour pouvoir réunir différents morceaux de code de façon suffisamment fiable.
— le reste de la spécification (contrats de fonctions, etc) est alors disponible en commentaires ou dans un document de spécification séparé.

Une spécification en OCaml sera décrite dans un fichier `.mli`. Une implantation (du code, donc) sera décrite dans un fichier `.ml` de même nom racine. Ceci n'est qu'un cas particulier de la notion plus générale de module qu'on retrouve en OCaml (et que l'on abordera dans la dernière partie du cours).

6 Exercices

▷ **Exercice 1**
— *Donner une fonction qui renvoie le i-ème élément d'un triplet.*
— *Écrire des opérations arithmétiques usuelles sur des fractions.*

▷ **Exercice 2**
— *Donner le type des définitions suivantes :*
```
let def1 x = if x >= 0 then 2*x else -2*x
let def2 x = x*.x
```
— *Donner une autre écriture, à l'aide de la composition de fonctions, de* `def1`*.*
— *Trouver une expression de type* `int -> bool`*.*

▷ **Exercice 3**

— *Donner des fonctions des types suivants :*

```
int -> int -> bool
int -> (int -> int)
(int -> int) -> int
int -> int * int
'a -> int
'a -> 'b -> 'a
int -> 'a
'a -> 'b
```

— *Donner le type des fonctions suivantes :*

```
# let id x = x;;

# let apply f x = f x;;

# let double f x = f (f x);;
```

▷ **Exercice 4**

— *Quel est le type de* `fst` *et* `snd` *?*

— *Donner une autre fonction polymorphe*

— *Donner le type des expressions suivantes :*

```
((1, "toto"), (2, "titi"))
(true, (1, (32, 4.5)))
```

— *Évaluer les expressions suivantes :*

```
fst (snd ((0, "toto"), (1.1, "titi")))
fst (snd ((1, 2), 3))
fst snd (1, (2, 3))
```

▷ **Exercice 5** *Écrire la fonction* `fst`.