Récursivité, Typage, Filtrage

11 septembre 2023

Plan

Remarques sur le TP

Déclarations qui se suivent

Deux déclarations peuvent se succéder.

```
1 let a = 1
2 let b = 2
```

Déclarations, puis expression

On ne peut pas mettre une expression après une déclaration.

```
1 let a = 1
2 a
```

Déclarations, puis expression

On ne peut pas mettre une expression après une déclaration.

```
1 let a = 1
2 a
```

OCaml interpréte l'espace entre `1` et `a` comme étant une application.

Plusieurs solutions

- Séparateur ;; :

```
let a = 1;;
a
```

- Variable locale :

```
let a = 1 in
a
```

Parenthèses en OCaml

```
1 f -1
```

Parenthèses en OCaml

```
f -1
```

```
f (-1)
```

Parenthèses en OCaml

```
1 let rec fact n = match n with
2 |0 -> 1
3 |_ -> n * (fact n - 1)
```

Parenthèses en OCaml

```
let rec fact n = match n with
|0 -> 1
|_ -> n * (fact n - 1)
```

```
let rec fact n = match n with
|0 -> 1
|_ -> n * (fact (n - 1))
```

Il *peut* y avoir trop de parenthèses dans une expression. On peut utiliser d'autres solutions pour rendre le code plus lisible :

- utilisation de variables locales intermédaires :

  ```
  let x1,y1 = couple in ajouter x1 y1 ;
  ```

- utilisation de `begin ... end` ;
- règles de priorité en OCaml.

Règles de priorité en OCaml

L'application de fonctions est prioritaire sur la plupart des opérateurs infixes que nous utiliserons cette année.

Exemple de priorité opérateur / application(1)

```
1 let rec fact n = match n with
2 |0 -> 1
3 |_ -> n * (fact (n - 1))
```

Exemple de priorité opérateur / application(1)

```
let rec fact n = match n with
|0 -> 1
|_ -> n * (fact (n - 1))
```

```
let rec fact n = match n with
|0 -> 1
|_ -> n * fact (n - 1)
```

Exemple de priorité opérateur / application(2)

```
1 ((fst couple1) -. (fst couple2))**2.
```

Exemple de priorité opérateur / application(2)

```
((fst couple1) -. (fst couple2))**2.
```

```
(fst couple1 -. fst couple2)**2.
```

L'application de fonction est associative à gauche

En l'absence de parenthèses, on applique les fonctions de gauche à droite.

```
let couple = 1, 2 in
let ajouter x y = x + y in
ajouter fst couple snd couple
```

L'application de fonction est associative à gauche

En l'absence de parenthèses, on applique les fonctions de gauche à droite.

```
let couple = 1, 2 in
let ajouter x y = x + y in
ajouter fst couple snd couple
```

```
let couple = 1, 2 in
let ajouter x y = x + y in
ajouter (fst couple) (snd couple)
```

Mot-clé when

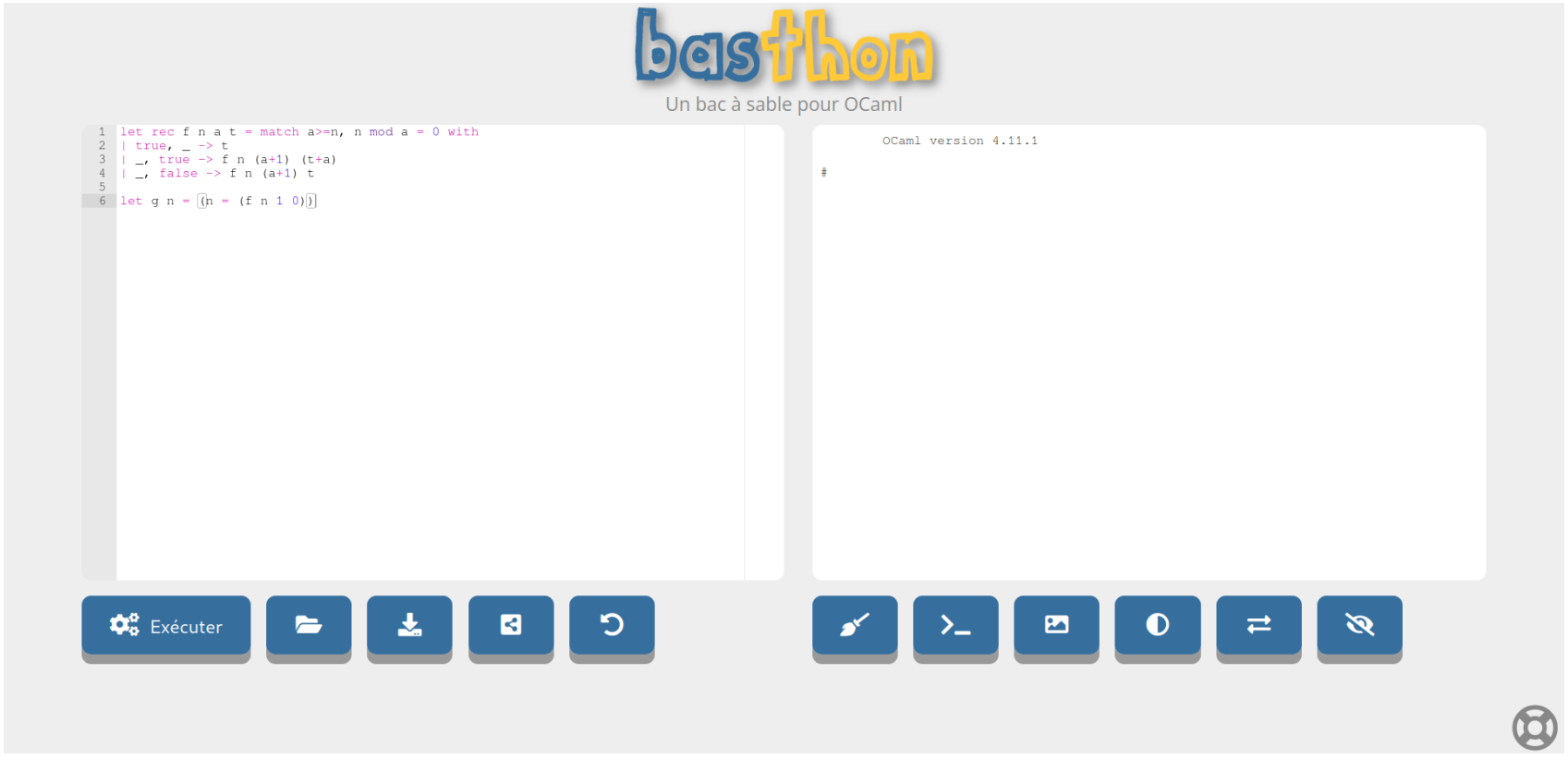
On peut spécifier le filtrage avec le mot-clef when.

```
let relu x = match x with
| _ when x>0. -> x
| _ -> 0.
```

Types de base

Nous reverrons plus en détail ces histoires de booléens, d'erreur de flottants, et de dépassement de mémoire des entiers dans un chapitre ultérieur.
(Mais il ne faut pas oublier d'utiliser les opérateurs spéciaux liés aux flottants : $+.$, $-.$, $*.$, $/.$ et $**.$)

Console Basthon



Installation d'OCaml

Les informations d'installation se trouve dans la documentation officielle [1] .
Vous aurez accés aux outils suivants :

- Console OCaml ;
- Compilateur OCaml.

1. https://v2.ocaml.org/docs/ install . fr . html

Compilation

Définition 1 : Compilation

La **compilation** est le processus de transformation d'un programme d'un langage vers un autre, usuellement de plus bas niveau, c'est-à-dire plus proche de la machine et donc plus facile à exécuter.
Le logiciel qui réalise la compilation est appelé **compilateur**.

Langage interprété, compilé

Étapes pour un langage interprété :

- J'écris mon code source `main.py` ;
- À chaque fois que j'ai besoin d'exécuter mon code, l'interpréteur l'exécute depuis le fichier source `python main.py`.

Langage interprété, compilé

Étapes pour un langage interprété :

- J'écris mon code source `main.py` ;
- À chaque fois que j'ai besoin d'exécuter mon code, l'interpréteur l'exécute depuis le fichier source `python main.py`.

Étapes pour un langage compilé :

- J'écris mon code source `main.ml` ;
- Je le compile vers un fichier exécutable adapté à ma machine : `ocamlopt main.ml -o main` ;
- À chaque fois que j'ai besoin d'exécuter mon programme, j'utilise directement le fichier exécutable : `./main`.

Récursivité

Récursivité

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot ((n-1)!) & \text{sinon.} \end{cases}$$

Récursivité

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot ((n-1)!) & \text{sinon.} \end{cases}$$

```
let rec fact n = match n with
| 0 -> 1
| n -> n * fact (n - 1)
```

Fonction récursive

Définition 2

Une fonction récursive est une fonction qui peut s'appeler elle-même.

Attention : c'est bien la fonction, et donc le programme qui peut être récursif. Un algorithme n'est pas nécessairement récursif en soi.

Plus facile de démontrer la correction du code

$$u_0 = 0$$
$$\forall n \geqslant 1 \; u_n = 2u_{n-1} + 1$$

```
let rec u n = match n with
| 0 -> 0
| n -> 2 * u (n - 1) + 1
```

Plus facile de démontrer la correction du code

$$u_0 = 0$$
$$\forall n \geqslant 1 \; u_n = 2u_{n-1} + 1$$

```
let rec u n = match n with
| 0 -> 0
| n -> 2 * u (n - 1) + 1
```

Par récurrence sur n, on peut démontrer que notre fonction termine ET a le résultat attendu.

Attention à votre cas de base

Comme pour une récurrence, il faut faire attention au cas de base :

```
let rec puissance x n = match n with
| 1 -> x
| _ -> x * puissance x (n-1)
```

```
puissance x 0
```

Attention à votre cas de base

Comme pour une récurrence, il faut faire attention au cas de base :

```
let rec puissance x n = match n with
| 1 -> x
| _ -> x * puissance x (n-1)
```

```
puissance x 0
```

```
let rec puissance x n = match n with
| 0 -> 1
| _ -> x * puissance x (n-1)
```

Définitions multiple

On peut définir plusieurs variables en même temps grâce au mot-clef `and` :

```
let a = 1
and b = 2 in
a + b
```

Définitions multiple

On peut définir plusieurs variables en même temps grâce au mot-clef `and` :

```
let a = 1
and b = 2 in
a + b
```

```
let a = 1 in
let b = 2 in
a + b
```

Récursivité croisée

L'utilité principale est de pouvoir définir des objets inter-dépendants.

```
let rec est_pair n = match n with
| 0 -> true
| _ -> est_impair (n-1)
and est_impair n = match n with
| 0 -> false
| _ -> est_pair (n-1)
```

Appels multiples

$$\forall 0 \leqslant k \leqslant n, \binom{n}{k} = \begin{cases} 1 \text{ si } k = 0 \text{ ou } k = n, \\ \binom{n-1}{k} + \binom{n-1}{k-1} \text{ sinon.} \end{cases}$$

```
let rec combin k n = match k with
|0 -> 1
|_ when k = n -> 1
|_ -> combin k (n-1) + combin (k-1) (n-1)
```

Programmation Fonctionnelle

Paradigme de programmation

Définition 3 : Paradigme de programmation

Un **paradigme de programmation** décrit la manière d'approcher la programmation face à un problème.

La plupart des langages permettent plusieurs paradigmes de programmation : le Python permet de la programmation impérative (séquence d'instructions qui modifient l'état du programme lors de l'exécution), de la programmation impérative structurée (structures et blocs conditionnels), de la programmation orientée objet, et de la programmation fonctionnelle.

Langage fonctionnel

Définition 4 : Programmation déclarative

Dans la programmation déclarative, les composants du programmes s'évaluent de manière indépendante du contexte.

Définition 5 : Programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui réalise la plupart des calculs par l'évaluation de fonctions.

Un **langage fonctionnel** est un langage qui encourage la programmation fonctionnelle.

OCaml et paradigme

OCaml est un langage fonctionnel : il permet de la programmation déclarative fonctionnelle.
Nous verrons au prochain chapitre qu'OCaml n'est pas un langage purement fonctionnel : il incorpore des outils de programmation impérative.

Applications d'OCaml

- Principalement dans la recherche et l'enseignement ;
- mais aussi dans l'industrie.

Type de donnée

Définition 6 : Type de donnée

Un **type de donnée**, ou **type** définit pour chaque donnée les différentes valeurs qu'elle peut prendre ainsi que les opérations qui sont autorisées avec cette valeur.

Définition 7 : Typage

Le typage est le procédé qui associe à chaque donnée un type.

Le typage peut se faire avant l'exécution (typage statique) ou durant l'exécution (typage dynamique).

Typage statique fort

- Le typage d'OCaml est statique : toute les vérifications de typage se font avant d'exécuter le code.
- Par ailleurs le typage d'OCaml est fort : il n'y a pas de conversion implicites et les types sont bien compartimentés.
- Ce typage est inféré : il n'y a pas besoin d'informations données par la personne qui programme pour faire le typage.

Type paramétré

Parfois OCaml ne peut pas déterminer exactement le type d'une fonction, dans ce cas il propose un type paramétré.

```
1 fst
```

```
 'a * 'b -> 'a
```

OCaml propose toujours le type le plus général qu'il trouve.

Type paramétré et fonctions en argument

```
1 let f x g = g x

'a -> ('a -> 'b) -> 'b
```

Fonction anonyme

Certaines fonctions prennent en argument d'autres fonctions.

```
let f g (x,y) = (g x, g y)
```

```
let g x = x + 1 in
f g (1, 2)
```

Fonction anonyme

Certaines fonctions prennent en argument d'autres fonctions.

```
let f g (x,y) = (g x, g y)
```

```
let g x = x + 1 in
f g (1, 2)
```

Le mot clef `fun` permet d'introduire des fonction anonymes :

```
f (fun x -> x + 1) (1, 2)
```

Cela est similaire au mot-clé lambda en python.

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Le mot-clé rec se met toujours au même endroit :

```
let rec fact =
    fun n -> match n with
    | 0 -> 0
    | _ -> n * fact (n - 1)
```

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Le mot-clé rec se met toujours au même endroit :

```
let rec fact =
    fun n -> match n with
    | 0 -> 0
    | _ -> n * fact (n - 1)
```

En règle générale, on peut se passer du mot-clé fun.

Cette semaine : TD sur les fonctions, la récursivité, et le typage.