Programmation fonctionnelle pour le web

I – Introduction et rappels

Guillaume Geoffroy
guillaume.geoffroy@irif.fr

UFR d'Informatique

L1 2023–2024

Introduction à l'UE *Programmation fonctionnelle pour le web*

Bienvenue en deuxième semestre de licence.

Cette UE est la suite d'*Introduction à la programmation fonctionnelle*.

Objectifs de l'UE

1. Poursuite de l'apprentissage de la programmation fonctionnelle.
   - Types algébriques.
   - Modules.
2. OCaml comme langage de développement.
   - Bibliothèque standard (notamment conteneurs).
   - Traits impératifs (données modifiables et interférences ; exceptions).
   - Écosystème : chaîne de compilation (notamment dune) ; gestionnaire de paquets (opam).
3. Écriture des clients web en OCaml.
   - Protocole HTTP(S) ; langage de données JSON ; bibliothèques OCaml.

Organisation des CM & maximiser l'efficacité de votre temps de travail

Chaque CM contient de nombreux exercices :

- À faire *pendant la séance*.
- À terminer et refaire (sans regarder la solution) *chaque semaine* avant le CM suivant.
  - Optimisation rapport note finale / temps de travail.
- À refaire avant le partiel et avant l'examen.

Organisation de l'UE

Planning

Chaque semaine : 2h de cours et 2h de travaux pratiques.

	Semaine 1	Semaine 12	Enseignants
Cours	22/01–26/01	29/04–03/05	G. Geoffroy
TP	29/01–02/02	29/04–03/05	C. Allain & A. Lancelot

Modalités de contrôle des connaissances

Partiel (début mars) et examen, respectivement 30% et 70% de la note finale.

Prérequis logiciels pour les travaux pratiques

- Début du semestre : sur le web via *LearnOCaml*.
  - Comme pour *Introduction à la programmation fonctionnelle*.
  - Rien à installer sur la machine locale.
- Fin du semestre : utilisation de l'écosystème OCaml réel.
  - Nécessaire pour faire des travaux pratiques web intéressants.
  - L'équipe enseignante peut vous aider à installer OCaml et les outils.
  - **Pas d'urgence immédiate**.

`https://ocaml.org/learn/tutorials/up_and_running.html`

Le reste de cette première séance

- Révision des bases d'OCaml acquises au premier semestre.

Plan

Plan

Le langage OCaml

Un langage de programmation...

- *fonctionnel* : les applications de fonctions sont prépondérantes,
- *typé* : rejette les programmes mal typés avant l'exécution,
- *polyvalent* : utilisé dans les domaines universitaire et industriel.

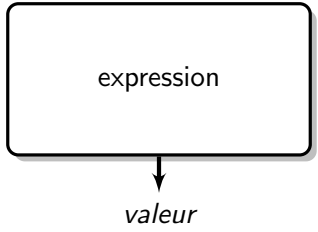
Le langage OCaml

Un langage de programmation...

- *fonctionnel* : les applications de fonctions sont prépondérantes,
- *typé* : rejette les programmes mal typés avant l'exécution,
- *polyvalent* : utilisé dans les domaines universitaire et industriel.

Expressions et valeurs

Un programme OCaml est principalement formé d'expressions.

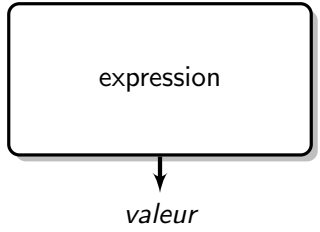


Les expressions et les valeurs

- Une *expression* est un fragment de code qui calcule une *valeur*.
  - On dit que l'expression s'*évalue* vers cette valeur.
- Une *valeur* est le résultat inerte d'un calcul terminé.
  - Par exemple : un entier, une chaîne de caractère...

Expressions et valeurs

Un programme OCaml est principalement formé d'expressions.



Les expressions et les valeurs

- Une *expression* est un fragment de code qui calcule une *valeur*.
  - On dit que l'expression s'*évalue* vers cette valeur.
- Une *valeur* est le résultat inerte d'un calcul terminé.
  - Par exemple : un entier, une chaîne de caractère...

Question

Pouvez-vous citer d'autres exemples de valeurs ?

Expressions, valeurs et variables

Question

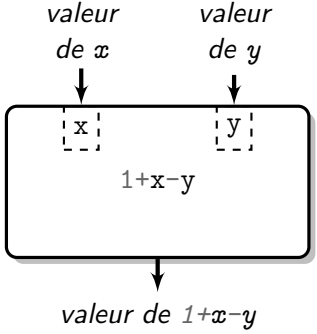
Vers quoi s'évaluent les expressions `42`, `1+1` et `1+x-y` ?

Expressions, valeurs et variables

Question

Vers quoi s'évaluent les expressions `42`, `1+1` et `1+x-y` ?

Pour déterminer la valeur de `1+x-y`, on doit connaître la valeur de `x` et `y`.

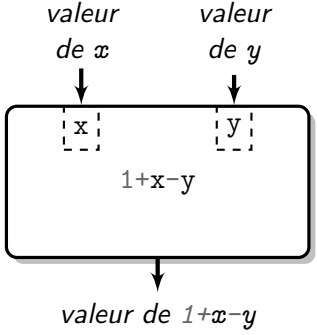


Expressions, valeurs et variables

Question

Vers quoi s'évaluent les expressions `42`, `1+1` et `1+x-y` ?

Pour déterminer la valeur de `1+x-y`, on doit connaître la valeur de `x` et `y`.



Les variables : de simples liens !

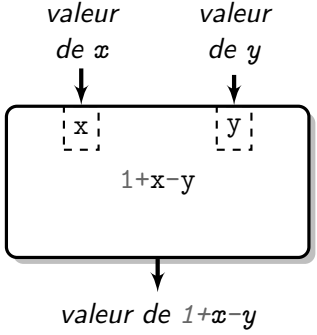
- Des noms pour des valeurs déjà calculées.
- On dit qu'une variable est *liée* à sa valeur.
- Peuvent être *redéfinies*, pas *modifiées*.
  - (On reviendra sur cette distinction.)

Expressions, valeurs et variables

Question

Vers quoi s'évaluent les expressions `42`, `1+1` et `1+x-y` ?

Pour déterminer la valeur de `1+x-y`, on doit connaître la valeur de `x` et `y`.



Les variables : de simples liens !

- Des noms pour des valeurs déjà calculées.
- On dit qu'une variable est *liée* à sa valeur.
- Peuvent être *redéfinies*, pas *modifiées*.
  - (On reviendra sur cette distinction.)

Attention aux confusions

Les variables d'OCaml ne sont pas comparables à celles de Python ou Java.

Les expressions et leurs types

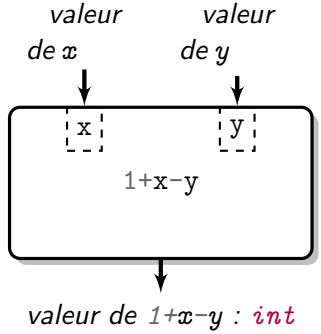
Toute expression e se voit assigner un type, petite formule logique qui...

- classifie les valeurs qui peuvent résulter de l'évaluation de e ;
- est déterminé par des règles dépendant uniquement de la forme de e.

Les expressions et leurs types

Toute expression e se voit assigner un type, petite formule logique qui...

- classifie les valeurs qui peuvent résulter de l'évaluation de e ;
- est déterminé par des règles dépendant uniquement de la forme de e.



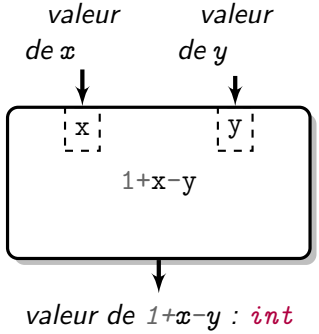
Les types comme contrats

Je vous garantis que l'expression s'évalue vers un entier...

Les expressions et leurs types

Toute expression e se voit assigner un type, petite formule logique qui...

- classifie les valeurs qui peuvent résulter de l'évaluation de e ;
- est déterminé par des règles dépendant uniquement de la forme de e.




```
let x = "coucou" in
let y = fun x -> x in
1+x-y (* valeur entière ? *)
```

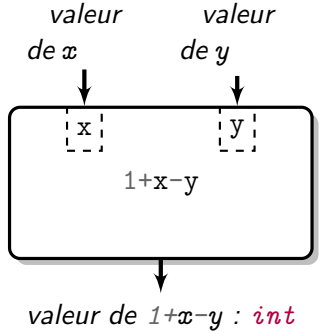
Les types comme contrats

Je vous garantis que l'expression s'évalue vers un entier...

Les expressions et leurs types

Toute expression e se voit assigner un type, petite formule logique qui...

- classifie les valeurs qui peuvent résulter de l'évaluation de e ;
- est déterminé par des règles dépendant uniquement de la forme de e.




```
let x = "coucou" in
let y = fun x -> x in
1+x-y (* valeur entière ? *)
(* Non, mais l'expression
   est mal typée ! *)
```

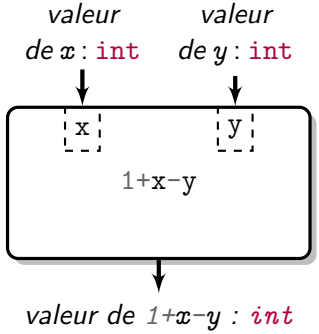
Les types comme contrats

Je vous garantis que l'expression s'évalue vers un entier...

Les expressions et leurs types

Toute expression e se voit assigner un type, petite formule logique qui...

- classifie les valeurs qui peuvent résulter de l'évaluation de e ;
- est déterminé par des règles dépendant uniquement de la forme de e.




```
let x = "coucou" in
let y = fun x -> x in
1+x-y (* valeur entière ? *)
(* Non, mais l'expression
   est mal typée ! *)
```

Les types comme contrats

Je vous garantis que l'expression s'évalue vers un entier... si vous me garantissez que les variables `x` et `y` sont liées à des valeurs entières !

Changer le monde

Un peu de terminologie

- Le *monde*, c'est tout ce qui n'appartient pas à notre programme.
  - La mémoire, la sortie standard, l'écran, le disque, le réseau...
- Une expression peut interagir avec le monde, en plus de son résultat.
  - Cette interaction est appelée l'*effet* de l'expression.
- Les types ne parlent pas des effets, uniquement des valeurs calculées.

Question

Pouvez-vous donner des exemples d'expressions qui ont un effet ?

Changer le monde

Un peu de terminologie

- Le *monde*, c'est tout ce qui n'appartient pas à notre programme.
  - La mémoire, la sortie standard, l'écran, le disque, le réseau...
- Une expression peut interagir avec le monde, en plus de son résultat.
  - Cette interaction est appelée l'*effet* de l'expression.
- Les types ne parlent pas des effets, uniquement des valeurs calculées.

Question

Pouvez-vous donner des exemples d'expressions qui ont un effet ?

Quelques exemples d'expressions qui interagissent avec le monde :

- `print_endline "Hello world!"`
- `1 + (print_endline "Hello world!"; 2)`
- `"Bonjour " ^ (read_line ()) ^ ", comment allez-vous ?"`

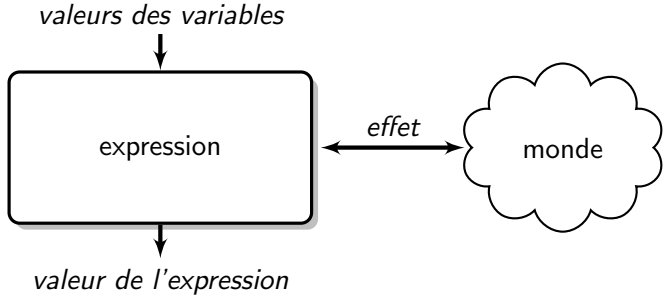
On en verra des effets de bien d'autres natures ce semestre.

Résumé

L'évaluation d'une expression...

- dépend des *valeurs* liées à ses variables ;
- produit, si elle termine, un résultat qui est une valeur du type attendu ;
- a éventuellement une interaction avec le monde extérieur, son *effet*.

Une représentation métaphorique :



On va maintenant réviser les construction vues au premier semestre.

Plan

Au programme

On va rappeler syntaxe, évaluation et typage des constructions suivantes :

1. les constantes littérales,
2. les définitions locales,
3. les fonctions,
4. les fonctions récursives,
5. les types énumérés, les listes, et le filtrage.

Préparez vos questions !

C'est un bon moment pour poser vos éventuelles questions sur les points qui restent confus ou délicats après le premier semestre.

Constantes littérales

`true`, `false`, `42`, `-5`, `3.14`, `42.0`, `"toto"`...

Évaluation

Une constante littérale s'évalue vers elle-même.

Typage

Chaque constante a un type dit *de base* fixé : `bool`/`int`/`float`/`string` pour les booléens/entiers/flottants/chaînes de caractères.

Conditionnelles

```
if e1 then e2 else e3
```

Évaluation

1. J'évalue e1 et obtiens une valeur booléenne b.
2. Si b est `true`, mon résultat est celui de l'évaluation de e2.
3. Si b est `false`, mon résultat est celui de l'évaluation de e3.

Typage

1. e1 doit être de type `bool`,
2. e2 et e3 doivent être de même type t quelconque,
3. `if e1 then e2 else e3` est alors de type t.

Exercices

Exercice 1

Donner un type à `x` qui rende chaque expression bien typée si possible.

1. `if x >= 0 then x else - x`
2. `if "coucou " ^ x then 12 else 0`
3. `if x <= 0 && x > 0 then 5 else "ohoh"`

On dit que deux expressions sont *équivalentes* si elles s'évaluent toujours vers la même valeur et avec le même effet.

Exercice 2

Donner les expressions équivalentes.

1. `if x then 42 else 2`
2. `2 + if x then 40 else 0`
3. `if x then (print_endline "Hello!"; 42) else 2`
4. `if x then 42 else (print_endline "Hello!"; 2)`

Définitions locales

```
let x = e1 in e2
```

Évaluation

1. J'évalue `e1` vers une valeur `v1`.
2. Je remplace `x` par `v1` dans `e2`.
3. J'évalue l'expression obtenue ; mon résultat est son résultat.

Typage

1. `e1` doit être d'un type `t1` quelconque,
2. `e2` doit être d'un type quelconque `t2` mais sous l'hypothèse que toutes ses utilisations de `x` soient de type `t1`,
3. `let x = e1 in e2` est alors de type `t2`.

Exercices

Exercice 3

Donner les expressions équivalentes.

1. `let x = 3 in let y = x + 1 in x - y`
2. `let y = 3 in let x = y + 1 in x - y`
3. `let y = 3 in let x = y + 1 in 2 * x`
4. `let x = 3 in let x = x + 1 in 2 * x`

Exercices

Exercice 3

Donner les expressions équivalentes.

1. `let x = 3 in let y = x + 1 in x - y`
2. `let y = 3 in let x = y + 1 in x - y`
3. `let y = 3 in let x = y + 1 in 2 * x`
4. `let x = 3 in let x = x + 1 in 2 * x`

Remarque

Les variables liées sont comme les variables muettes en mathématiques : on peut toujours les renommer sans changer aucun résultat.

Les fonctions

`fun x -> e` et `e1 e2`

Évaluation

- `fun x -> e` s'évalue vers elle-même : les fonctions sont des valeurs !
- L'évaluation de `e1 e2` ressemble assez à celle du `let`.
  1. J'évalue `e2` vers une valeur `v2`.
  2. J'évalue `e1` vers une fonction `fun x -> e1'`.
  3. Je remplace `x` par `v2` dans `e1'`.
  4. J'évalue l'expression obtenue, son résultat est mon résultat.

Typage

- Si `e` est de type `t2` en utilisant `x` au type `t1`, alors `fun x -> e` est de type `t1 -> t2`.
- Pour l'application `e1 e2` :
  1. `e1` doit être de type `t1 -> t2` pour `t1` et `t2` quelconques,
  2. `e2` doit être de type `t1`,
  3. `e1 e2` est alors de type `t2`.

Les fonctions : sucre syntaxique

- On facilite l'utilisation des fonctions à plusieurs arguments.

$$\texttt{fun x1 ... xN -> e} \triangleq \texttt{fun x1 -> ... -> fun xN -> e}$$

$$\texttt{e1 e2 ... eN} \triangleq \texttt{((e1 e2) ... eN)}$$

$$\texttt{t1 -> t2 -> t3} \triangleq \texttt{t1 -> (t2 -> t3)}$$

- On facilite les définitions locales de fonction.

$$\texttt{let f x1 ... xN = e} \triangleq \texttt{let f = fun x1 ... xN -> e}$$

- On facilite l'utilisation d'opérateurs infixes comme fonctions.

$$\texttt{(+)} \triangleq \texttt{fun x y -> x + y}$$

Les fonctions : sucre syntaxique

- On facilite l'utilisation des fonctions à plusieurs arguments.

$$\texttt{fun x1 ... xN -> e} \triangleq \texttt{fun x1 -> ... -> fun xN -> e}$$

$$\texttt{e1 e2 ... eN} \triangleq \texttt{((e1 e2) ... eN)}$$

$$\texttt{t1 -> t2 -> t3} \triangleq \texttt{t1 -> (t2 -> t3)}$$

- On facilite les définitions locales de fonction.

$$\texttt{let f x1 ... xN = e} \triangleq \texttt{let f = fun x1 ... xN -> e}$$

- On facilite l'utilisation d'opérateurs infixes comme fonctions.

$$\texttt{(+)} \triangleq \texttt{fun x y -> x + y}$$

Question

Écrire sans sucre syntaxique la définition suivante.

```
let f g x y = g x y - 1 in let g f = f (-) 2 in g f 3
```

Est-elle bien typée ? Si oui, donner le résultat de son évaluation.

Exercice

Exercice

Écrire la règle de typage de `e1 e2 e3` dans le même style que les explications des transparents précédents. De même pour `fun x y -> e`.

Les fonctions récursives (pour la culture : ne pas essayer de le retenir !)

```
let rec f x = e1 in e2
```

Évaluation

1. Je remplace `f` par `fun x -> let rec f x = e1 in e1` dans `e2`.
2. J'évalue l'expression obtenue, son résultat est mon résultat.

Typage

1. `e1` doit être de type `t2` en utilisant `x` au type `t1` et `f` au type `t1 -> t2`,
2. `e2` doit être de type `t` en utilisant `f` au type `t1 -> t2`,
3. `let rec f x = e1 in e2` est alors de type `t`.

Les définitions de types

- Les programmes peuvent aussi contenir des définitions de types.
- On peut définir :

Les définitions de types

- Les programmes peuvent aussi contenir des définitions de types.
- On peut définir :
  - un synonyme (alias) pour un type existant ;

```
type int_pair = int * int
```

Les définitions de types

- Les programmes peuvent aussi contenir des définitions de types.
- On peut définir :
  - un synonyme (alias) pour un type existant ;
  - un nouveau type enregistrement, en énumérant ses champs ;

```
type int_pair = int * int
type person = { firstname : string;
                lastname : string;
                age : int; }
```

Les définitions de types

- Les programmes peuvent aussi contenir des définitions de types.
- On peut définir :
  - un synonyme (alias) pour un type existant ;
  - un nouveau type enregistrement, en énumérant ses champs ;
  - un nouveau type somme, en énumérant ses constructeurs.

```
type int_pair = int * int
type person = { firstname : string;
                lastname : string;
                age : int; }
type figure = Square | Circle | Triangle
```

Les définitions de types

- Les programmes peuvent aussi contenir des définitions de types.
- On peut définir :
  - un synonyme (alias) pour un type existant ;
  - un nouveau type enregistrement, en énumérant ses champs ;
  - un nouveau type somme, en énumérant ses constructeurs.
  - les types enregistrement et somme peuvent être récursifs.

```
type int_pair = int * int
type person = { firstname : string;
                lastname : string;
                age : int; }
type figure = Square | Circle | Triangle
type int_list =
  | Nil
  | Cons of int * int_list
```

Les définitions de types paramétrés

- On peut définir des types *paramétrés* par d'autres types.
- Pour le paramétrage, on utilise des *variables de types*.
  - Leurs noms commencent par une apostrophe, p. ex. `'a`, `'b` ou `'toto`.

Les définitions de types paramétrés

- On peut définir des types *paramétrés* par d'autres types.
- Pour le paramétrage, on utilise des *variables de types*.
  - Leurs noms commencent par une apostrophe, p. ex. `'a`, `'b` ou `'toto`.

```
type 'a printer = 'a -> string
```

Les définitions de types paramétrés

- On peut définir des types *paramétrés* par d'autres types.
- Pour le paramétrage, on utilise des *variables de types*.
  - Leurs noms commencent par une apostrophe, p. ex. `'a`, `'b` ou `'toto`.

```
type 'a printer = 'a -> string
type ('a, 'b) pair = { l : 'a; r : 'b; }
```

Les définitions de types paramétrés

- On peut définir des types *paramétrés* par d'autres types.
- Pour le paramétrage, on utilise des *variables de types*.
  - Leurs noms commencent par une apostrophe, p. ex. `'a`, `'b` ou `'toto`.

```
type 'a printer = 'a -> string
type ('a, 'b) pair = { l : 'a; r : 'b; }
type 'a list =
  | []
  | (::) of 'a * 'a list
```

Le filtrage

```
match e with p1 -> e1 | ... | pN -> eN
```

Évaluation

1. J'évalue `e` vers une valeur `v`.
2. Si `p1` filtre `v`, alors j'évalue `e1` dans l'environnement obtenu.
3. Sinon, j'essaie les branches suivantes jusqu'à `pN -> eN` incluse.
4. Si aucun motif ne filtre `v`, je lève une exception.

Typage

1. `e` doit être de type `t` quelconque,
2. `p1` doit filtrer des valeurs de type `t` et `e1` doit être de type `t'` en utilisant les variables liées par `p1` aux bons types,
3. ... et ainsi de suite jusqu'à `pN` et `eN`,
4. `match e with p1 -> e1 | ... | pN -> eN` est alors de type `t'`.

Le filtrage

```
match e with p1 -> e1 | ... | pN -> eN
```

Évaluation

1. J'évalue `e` vers une valeur `v`.
2. Si `p1` filtre `v`, alors j'évalue `e1` dans l'environnement obtenu.
3. Sinon, j'essaie les branches suivantes jusqu'à `pN -> eN` incluse.
4. Si aucun motif ne filtre `v`, je lève une exception.

Typage

1. `e` doit être de type `t` quelconque,
2. `p1` doit filtrer des valeurs de type `t` et `e1` doit être de type `t'` en utilisant les variables liées par `p1` aux bons types,
3. ... et ainsi de suite jusqu'à `pN` et `eN`,
4. `match e with p1 -> e1 | ... | pN -> eN` est alors de type `t'`.

Le filtrage

```
match e with p1 -> e1 | ... | pN -> eN
```

Évaluation

1. J'évalue `e` vers une valeur `v`.
2. Si `p1` filtre `v`, alors j'évalue `e1` dans l'environnement obtenu.
3. Sinon, j'essaie les branches suivantes jusqu'à `pN -> eN` incluse.
4. Si aucun motif ne filtre `v`, je lève une exception.

Typage

1. `e` doit être de type `t` quelconque,
2. `p1` doit filtrer des valeurs de type `t` et `e1` doit être de type `t'` en utilisant les variables liées par `p1` aux bons types,
3. ... et ainsi de suite jusqu'à `pN` et `eN`,
4. `match e with p1 -> e1 | ... | pN -> eN` est alors de type `t'`.

Les motifs

Chaque motif :

- décrit un ensemble de valeurs ; on dit qu'il les *filtre*,
- il lie aussi un ensemble de variables.

Motif	**Valeurs filtrées**	**Variables liées**
`_`	Toutes	Aucune
`x`	Toutes	`x`
`(p1, ..., pN)`	`(v1, ..., vN)` où `vI` filtre `pI`	Celles liées par les `pI`
`C (p1, ..., pN)`	`C (v1, ..., vN)` où `vI` filtre `pI`	idem
`p1 | p2`	celles filtrées par `p1` ou `p2`	`p1` et `p2` doivent lier les mêmes variables

Question

Dire quelles valeurs sont liées à quelles variables dans `e`.

1. `match [(1, "toto"); (3, "")] with (_, y) :: z -> e`
2. `match [(1, "toto"); (3, "")] with [c; z] -> e`

Plan

Le typage en OCaml

L'inférence de types

- OCaml détermine les types avant l'exécution du programme.
- Il calcule le type *le plus général* de chaque expression.
  - "Le plus général" au sens où tous les autres en sont des cas particuliers.
  - Ce type s'appelle le type *principal* de l'expression.
- Ce principe justifie l'ajout du *polymorphisme*.

Question

Donnez le type le plus général des expressions suivantes.

1. `fun x -> x`
2. `fun f x -> f (x + 1)`
3. `let c f g = fun x -> g (f x) in fun h -> c h ((+) 1)`

Exercice

Questions

Donner le type le plus général des programmes suivants.

```
let rec fold_right f xs acc =
  match xs with
  | [] -> acc
  | x :: xs -> f x (fold_right f xs acc)

let transform f xs =
  fold_right
    (fun x ys -> match f x with None -> ys
                              | Some y -> y :: ys)
    xs []
```

Plan

Conclusion

Bilan de la séance

- On a survolé tout le premier semestre de manière systématique.
- L'apprentissage d'un langage de programmation passe par la pratique.
  - Programmer permet d'internaliser les règles qui gouvernent le langage.

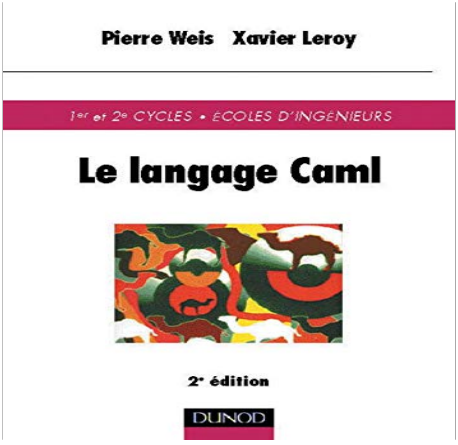
Les prochaines séances

- De nouvelles constructions, des structures de données !
  - Exploration des possibilités ouvertes par les types sommes (arbres).
  - Tableaux, références, champs modifiables.
- Une application à la programmation web en deuxième partie.

Deux livres de référence

http://programmer-avec-ocaml.lri.fr

- Moderne et récent.
- Disponible à la bibliothèque.
- Introduit l'algorithmique.



https://caml.inria.fr/pub/distrib/books/llc.pdf

- Plus ancien, pré-**O**Caml.
- Gratuit en ligne.
- Très bons exemples avancés.