Programmation fonctionnelle avancée

Notes de cours

Cours 1

14 Septembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Calendrier

Les TD commencent la semaine du 21 Septembre

Au total : 9 cours de 2h, 6 TD de 3h et 3 TP de 3h

Contrôle des connaissances :

- partiel : semaine du 26 octobre
- examen : semaine du 14 décembre

Retrouver toutes ces informations (et d'autres) sur le web

`http://www.lri.fr/~conchon/PFA/`

Les listes

Séquences d'entiers

On peut utiliser un type somme pour représenter des séquences (non bornées) d'entiers :

```
# type int_list = Nil | Cons of int * int_list;;
```

- `Nil` représente la séquence vide
- `Cons(x,l)` est la séquence dont le premier élément (on dit aussi la tête) est `x` et la suite est la séquence `l`

Par exemple, la séquence `4;1;5;8;1` est représentée par la valeur :

```
# Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil)))));;
- : int_list = Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil)))))
```

Le type `int list`

Le type des séquences est prédéfini en OCaml et ses éléments se notent avec une syntaxe spéciale

- `Cons` se note `::` et est infixe
- `Nil` se note `[]`

Par exemple, la séquence `4;1;5;8;1` est représentée par :

```
# 4::1::5::8::1::[];;
- : int list = [4;1;5;8;1]
```

On peut aussi directement utiliser la notation `[e1;e2;...;en]`

```
# [4;1;5;8;1];;
- : int list = [4;1;5;8;1]
```

ou faire un mélange des deux notations :

```
# 4::1::[5;8;1];;
- : int list = [4;1;5;8;1]
```

Des listes de types quelconques

OCaml permet de définir des listes dont les éléments peuvent être autre chose que des entiers :

```
# ['c';'a';'m';'l'];;
- : char list = ['c';'a';'m';'l']

# [["des";"listes"];["de";"listes"]];;
- : string list list = [["des";"listes"];["de";"listes"]]
```

Mais il n'est pas possible de construire une liste d'éléments de types différents :

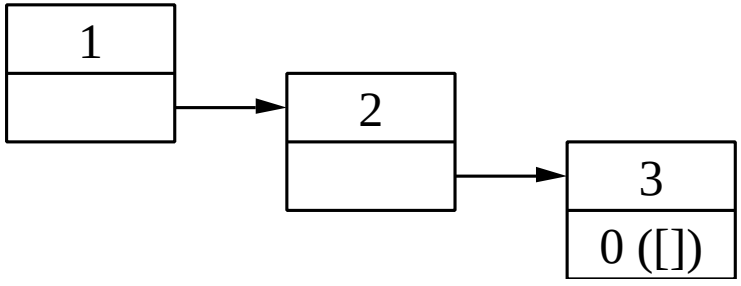
```
# [10;'a';4];;
      ---
This expression has type char but is here used
with type int
```

Listes chaînées

Les listes prédéfinies en OCaml correspondent exactement aux listes chaînées définies habituellement en C par le type suivant

```
typedef struct list{
  int elt;
  struct list* suivant;
};
```

La représentation mémoire de ces listes correspond à un chaînage de blocs mémoire, par exemple, la liste `[1; 2; 3]` correspond à :



Accès aux éléments d'une liste

On accède aux éléments d'une liste à l'aide des fonctions prédéfinies `List.hd` et `List.tl`

```
# List.hd [3;6;1;2];;
- : int = 3

# List.tl [3;6;1;2];;
- : int list = [6;1;2];;
```

`List.hd` et `List.tl` échouent sur une liste vide

```
# List.hd [];;
Exception: Failure "hd".

# List.tl [];;
Exception: Failure "tl".
```

Fonctions sur les listes

Définitions de fonctions sur les listes

La définition des fonctions sur les listes prennent généralement la forme d'une définition à deux cas :

- le cas où liste est vide
- le cas où elle ne l'est pas

Pour cette raison, il est plus agréable de réaliser cette analyse par cas avec du filtrage :

```
# let rec f l =
   match l with
     [] -> ...
   | x::s -> ...
```

la fonction `zeros`

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>
```

la fonction `zeros`

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true
```

la fonction `zeros`

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true

# zeros [0;0;0];;
- : bool = true
```

la fonction `zeros`

La fonction `zeros` vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie `true` si la liste est vide)

```
# let rec zeros l =
   match l with
     [] -> true
   | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>

# zeros [];;
- : bool = true

# zeros [0;0;0];;
- : bool = true

# zeros [0;1;0];;
- : bool = false
```

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

`zeros [0;0;0]`

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{rcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], \texttt{x} = 0, \texttt{s} = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

$$
\begin{array}{lcl}
 & & \texttt{zeros [0;0;0]} \\
[0;0;0] \neq [], \texttt{x} = 0, \texttt{s} = [0;0] & \Rightarrow & \texttt{0=0 \&\& zeros [0;0]} \\
 & = & \texttt{zeros [0;0]} \\
[0;0] \neq [], \texttt{x} = 0, \texttt{s} = [0] & \Rightarrow & \texttt{0=0 \&\& zeros [0]} \\
 & = & \texttt{zeros [0]}
\end{array}
$$

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

		`zeros [0;0;0]`
$[0;0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0;0]$	$\Rightarrow$	`0=0 && zeros [0;0]`
	$=$	`zeros [0;0]`
$[0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0]$	$\Rightarrow$	`0=0 && zeros [0]`
	$=$	`zeros [0]`
$[0] \neq [], \mathtt{x} = 0, \mathtt{s} = []$	$\Rightarrow$	`0=0 && zeros []`
	$=$	`zeros []`

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

		`zeros [0;0;0]`
$[0;0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0;0]$	$\Rightarrow$	`0=0 && zeros [0;0]`
	$=$	`zeros [0;0]`
$[0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0]$	$\Rightarrow$	`0=0 && zeros [0]`
	$=$	`zeros [0]`
$[0] \neq [], \mathtt{x} = 0, \mathtt{s} = []$	$\Rightarrow$	`0=0 && zeros []`
	$=$	`zeros []`
$[] = []$	$\Rightarrow$	`true`

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

		`zeros [0;0;0]`
$[0;0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0;0]$	$\Rightarrow$	`0=0 && zeros [0;0]`
	$=$	`zeros [0;0]`
$[0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0]$	$\Rightarrow$	`0=0 && zeros [0]`
	$=$	`zeros [0]`
$[0] \neq [], \mathtt{x} = 0, \mathtt{s} = []$	$\Rightarrow$	`0=0 && zeros []`
	$=$	`zeros []`
$[] = []$	$\Rightarrow$	`true`

Évaluation de `zeros [0;1;0]`

`zeros [0;1;0]`

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

		`zeros [0;0;0]`
$[0;0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0;0]$	$\Rightarrow$	`0=0 && zeros [0;0]`
	$=$	`zeros [0;0]`
$[0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0]$	$\Rightarrow$	`0=0 && zeros [0]`
	$=$	`zeros [0]`
$[0] \neq [], \mathtt{x} = 0, \mathtt{s} = []$	$\Rightarrow$	`0=0 && zeros []`
	$=$	`zeros []`
$[] = []$	$\Rightarrow$	`true`

Évaluation de `zeros [0;1;0]`

		`zeros [0;1;0]`
$[0;1;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [1;0]$	$\Rightarrow$	`0=0 && zeros [1;0]`
	$=$	`zeros [1;0]`

Évaluation de la fonction zeros

Évaluation de `zeros [0;0;0]`

		`zeros [0;0;0]`
$[0;0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0;0]$	$\Rightarrow$	`0=0 && zeros [0;0]`
	$=$	`zeros [0;0]`
$[0;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [0]$	$\Rightarrow$	`0=0 && zeros [0]`
	$=$	`zeros [0]`
$[0] \neq [], \mathtt{x} = 0, \mathtt{s} = []$	$\Rightarrow$	`0=0 && zeros []`
	$=$	`zeros []`
$[] = []$	$\Rightarrow$	`true`

Évaluation de `zeros [0;1;0]`

		`zeros [0;1;0]`
$[0;1;0] \neq [], \mathtt{x} = 0, \mathtt{s} = [1;0]$	$\Rightarrow$	`0=0 && zeros [1;0]`
	$=$	`zeros [1;0]`
$[1;0] \neq [], \mathtt{x} = 1, \mathtt{s} = [0]$	$\Rightarrow$	`1=0 && zeros [0]`
	$=$	`false`

Recherche d'un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>
```

Recherche d'un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>

# recherche 4 [3;2;4;1];;
- : bool = true
```

Recherche d'un entier dans une liste

La fonction `recherche` détermine si un entier n figure bien dans une liste `l`

```
# let rec recherche n l =
   match l with
     [] -> false
   | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>

# recherche 4 [3;2;4;1];;
- : bool = true

# recherche 4 [1;2];;
- : bool = false
```

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

`recherche 4 [3;2;4;1]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$

```
recherche 4 [3;2;4;1]
⇒ 3=4 || recherche 4 [2;4;1]
= recherche 4 [2;4;1]
```

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], \mathtt{x} = 4, \mathtt{s} = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], \mathtt{x} = 4, \mathtt{s} = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de `recherche 4 [1;2]`

`recherche 4 [1;2]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], \mathtt{x} = 4, \mathtt{s} = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de `recherche 4 [1;2]`

		`recherche 4 [1;2]`
$[1;2] \neq [], \mathtt{x} = 1, \mathtt{s} = [2]$	$\Rightarrow$	`1=4 || recherche 4 [2]`
	$=$	`recherche 4 [2]`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], \mathtt{x} = 4, \mathtt{s} = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de `recherche 4 [1;2]`

		`recherche 4 [1;2]`
$[1;2] \neq [], \mathtt{x} = 1, \mathtt{s} = [2]$	$\Rightarrow$	`1=4 || recherche 4 [2]`
	$=$	`recherche 4 [2]`
$[2] \neq [], \mathtt{x} = 2, \mathtt{s} = []$	$\Rightarrow$	`2=4 || recherche 4 []`
	$=$	`recherche 4 []`

Évaluation de la fonction `recherche`

Évaluation de `recherche 4 [3;2;4;1]`

		`recherche 4 [3;2;4;1]`
$[3;2;4;1] \neq [], \mathtt{x} = 3, \mathtt{s} = [2;4;1]$	$\Rightarrow$	`3=4 || recherche 4 [2;4;1]`
	$=$	`recherche 4 [2;4;1]`
$[2;4;1] \neq [], \mathtt{x} = 2, \mathtt{s} = [4;1]$	$\Rightarrow$	`2=4 || recherche [4;1]`
	$=$	`recherche 4 [4;1]`
$[4;1] \neq [], \mathtt{x} = 4, \mathtt{s} = [1]$	$\Rightarrow$	`4=4 || recherche 4 [1]`
	$=$	`true`

Évaluation de `recherche 4 [1;2]`

		`recherche 4 [1;2]`
$[1;2] \neq [], \mathtt{x} = 1, \mathtt{s} = [2]$	$\Rightarrow$	`1=4 || recherche 4 [2]`
	$=$	`recherche 4 [2]`
$[2] \neq [], \mathtt{x} = 2, \mathtt{s} = []$	$\Rightarrow$	`2=4 || recherche 4 []`
	$=$	`recherche 4 []`
$[] = []$	$\Rightarrow$	`false`

Longueur d'une liste

La fonction `longueur` retourne la longueur d'une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Longueur d'une liste

La fonction `longueur` retourne la longueur d'une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Une version récursive terminale :

```
# let longueur l =
   let rec longrec acc l =
     match l with
      [] -> acc
    | _::s -> longrec (1+acc) s
   in
   longrec 0 l
```

Longueur d'une liste

La fonction `longueur` retourne la longueur d'une liste

```
# let rec longueur l =
   match l with
     [] -> 0
   | _::s -> 1 + (longueur s);;
```

Une version récursive terminale :

```
# let longueur l =
   let rec longrec acc l =
     match l with
      [] -> acc
    | _::s -> longrec (1+acc) s
   in
   longrec 0 l
```

Cette fonction est prédéfinie en OCaml : `List.length`

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$
\begin{aligned}
& \texttt{longueur [10;2;4]} \\
= \; & \texttt{longrec 0 [10;2;4]}
\end{aligned}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

$$
\begin{array}{rcl}
 & & \texttt{longueur [10;2;4]} \\
 & = & \texttt{longrec 0 [10;2;4]} \\
[10;2;4] \neq [], \texttt{s} = [2;4] & \Rightarrow & \texttt{longrec (1+0) [2;4]}
\end{array}
$$

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

		`longueur [10;2;4]`
	$=$	`longrec 0 [10;2;4]`
$[10;2;4] \neq [], \mathtt{s} = [2;4]$	$\Rightarrow$	`longrec (1+0) [2;4]`
$[2;4] \neq [], \mathtt{s} = [4]$	$\Rightarrow$	`longrec (1+1) [4]`

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

		`longueur [10;2;4]`
	$=$	`longrec 0 [10;2;4]`
$[10;2;4] \neq [], \mathtt{s} = [2;4]$	$\Rightarrow$	`longrec (1+0) [2;4]`
$[2;4] \neq [], \mathtt{s} = [4]$	$\Rightarrow$	`longrec (1+1) [4]`
$[4] \neq [], \mathtt{s} = []$	$\Rightarrow$	`longrec (1+2) []`

Évaluation de la fonction `longueur`

Évaluation de `longueur [10;2;4]`

		`longueur [10;2;4]`
	$=$	`longrec 0 [10;2;4]`
$[10;2;4] \neq [], \mathtt{s} = [2;4]$	$\Rightarrow$	`longrec (1+0) [2;4]`
$[2;4] \neq [], \mathtt{s} = [4]$	$\Rightarrow$	`longrec (1+1) [4]`
$[4] \neq [], \mathtt{s} = []$	$\Rightarrow$	`longrec (1+2) []`
$[] = []$	$\Rightarrow$	`3`

Polymorphisme

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

```
# longueur [[4.5;0.3;9.8];[];[3.2;1.8]];;
- : int = 3
```

Fonctions polymorphes (1/2)

Quel est le type de la fonction `longueur` ?

`longueur` doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```
# longueur [4;3;6;1;10];;
- : int = 5
```

. . . alors elle doit avoir le type suivant

```
val longueur : int list -> int
```

Mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple :

```
# longueur [[4.5;0.3;9.8];[];[3.2;1.8]];;
- : int = 3
```

. . . dans ce cas la fonction `longueur` devrait également avoir

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

```
val longueur : 'a list -> int
```

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

```
val longueur : 'a list -> int
```

'a (qui se lit “apostrophe a”, ou encore “alpha”) est une variable de type

Une variable de type veut dire n’importe quel type

Fonctions polymorphes (2/2)

- Les deux types précédents sont corrects
- La fonction `longueur` a une infinité de types

Le type inféré par OCaml est le plus général :

```
val longueur : 'a list -> int
```

'a (qui se lit “apostrophe a”, ou encore “alpha”) est une variable de type

Une variable de type veut dire n'importe quel type

Il faut donc lire le type de la fonction `longueur` comme suit :

“La fonction `longueur` prend en argument une **liste** – dont les éléments sont de n'importe quel type – et retourne un **entier**”

Types sommes polymorphes

Les type sommes peuvent aussi être polymorphes

```
type 'a option = None | Some of 'a
```

```
# None;;
- : 'a option = None
# let v = Some(3);;
val v : int option = Some(3)
```

Un autre type somme polymorphe bien connu.

```
type 'a liste = Vide | Cellule of 'a * 'a liste
```

```
# Vide;;
- : 'a liste = Vide
# let l = Cellule(3,Vide);;
val l : int liste = Cellule(3,Vide)
```

Types sommes polymorphes

Les types peuvent également contenir plusieurs variables de type

```
type ('a, 'b) t = A of 'a * 'b | B of int
```

```
# A(1.2, "r");;
- : (float, string) t = A(1.2, "r")

# A('a', 1);;
- : (char, int) t = A('a', 1)
```

Allocation mémoire

- tous les objets sont alloués dans une zone mémoire appelée tas (sauf les types de base et sans compter les nombreuses optimisations du compilateur)
- l'allocation mémoire est réalisée par un garbage collector (GC) ce qui permet une récupération automatique de la mémoire et une allocation efficace

Fonctions génériques sur les listes

concaténation de listes

La fonction `append` construit une nouvelle la liste en réunissant deux listes bout à bout

```
# let rec append l1 l2 =
   match l1 with
     [] -> l2
   | x::s -> x::(append s l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
```

concaténation de listes

La fonction `append` construit une nouvelle la liste en réunissant deux listes bout à bout

```
# let rec append l1 l2 =
   match l1 with
     [] -> l2
   | x::s -> x::(append s l2);;
val append : 'a list -> 'a list -> 'a list = <fun>

# append [2;5;1] [10;6;8;15];;
- : int list = [2;5;1;10;6;8;15]
```

- Cette fonction est prédéfinie en OCaml, il s'agit de `List.append`
- L'opérateur infixe `@` est un raccourci syntaxique pour cette fonction, on note `l1@l2` la concaténation de `l1` et `l2`

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

`append [1;2] [3;4]`

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

$$[1;2] \neq [], \texttt{x} = 1, \texttt{s} = [2] \quad \Rightarrow$$

```
append [1;2] [3;4]
1::(append [2] [3;4])
```

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

$$
\begin{array}{lcl}
 & & \texttt{append [1;2] [3;4]} \\
[1;2] \neq [], \texttt{x} = 1, \texttt{s} = [2] & \Rightarrow & \texttt{1::(append [2] [3;4])} \\
[2] \neq [], \texttt{x} = 2, \texttt{s} = [] & \Rightarrow & \texttt{1::2::(append [] [3;4])}
\end{array}
$$

Évaluation de `append`

Évaluation de `append [1;2] [3;4]`

		`append [1;2] [3;4]`
$[1;2] \neq [], \mathtt{x} = 1, \mathtt{s} = [2]$	$\Rightarrow$	`1::(append [2] [3;4])`
$[2] \neq [], \mathtt{x} = 2, \mathtt{s} = []$	$\Rightarrow$	`1::2::(append [] [3;4])`
$[] = []$	$\Rightarrow$	`1::2::[3;4]`
	$\Rightarrow$	`1::[2;3;4]`
	$\Rightarrow$	`[1;2;3;4]`

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
val rev_append : 'a list -> 'a list -> 'a list = <fun>
```

Concaténation rapide

- La fonction `append` n'est pas récursive terminale
- Si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | x :: s -> rev_append s (x :: l2)
val rev_append : 'a list -> 'a list -> 'a list = <fun>

# rev_append [4;2;6] [1;10;9;5];;
- : int list = [6; 2; 4; 1; 10; 9; 5]
```

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

Renverser une liste

La fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

```
# let rev l = rev_append l [];;
val rev : 'a list -> 'a list = <fun>

# rev [4;2;6;1];;
- : int list = [1; 6; 2; 4]
```

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

```
    rev [1;2;3]
=   rev_append [1;2;3] []
```

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$
\begin{array}{lcl}
 & & \texttt{rev [1;2;3]} \\
 & = & \texttt{rev\_append [1;2;3] []} \\
[1;2;3] \neq [], \mathtt{x} = 1, \mathtt{s} = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\
 & = & \texttt{rev\_append [2;3] [1]}
\end{array}
$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

$$
\begin{array}{lcl}
 & & \texttt{rev [1;2;3]} \\
 & = & \texttt{rev\_append [1;2;3] []} \\
[1;2;3] \neq [], \mathtt{x} = 1, \mathtt{s} = [2;3] & \Rightarrow & \texttt{rev\_append [2;3] (1::[])} \\
 & = & \texttt{rev\_append [2;3] [1]} \\
[2;3] \neq [], \mathtt{x} = 2, \mathtt{s} = [3] & \Rightarrow & \texttt{rev\_append [3] (2::[1])} \\
 & = & \texttt{rev\_append [3] [2;1]}
\end{array}
$$

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

		`rev [1;2;3]`
	$=$	`rev_append [1;2;3] []`
$[1;2;3] \neq [], \mathtt{x} = 1, \mathtt{s} = [2;3]$	$\Rightarrow$	`rev_append [2;3] (1::[])`
	$=$	`rev_append [2;3] [1]`
$[2;3] \neq [], \mathtt{x} = 2, \mathtt{s} = [3]$	$\Rightarrow$	`rev_append [3] (2::[1])`
	$=$	`rev_append [3] [2;1]`
$[3] \neq [], \mathtt{x} = 3, \mathtt{s} = []$	$\Rightarrow$	`rev_append [] (3::[2;1])`
	$=$	`rev_append [] [3;2;1]`

Évaluation de `rev`

Évaluation de `rev [1;2;3]`

		`rev [1;2;3]`
	$=$	`rev_append [1;2;3] []`
$[1;2;3] \neq [], \mathtt{x} = 1, \mathtt{s} = [2;3]$	$\Rightarrow$	`rev_append [2;3] (1::[])`
	$=$	`rev_append [2;3] [1]`
$[2;3] \neq [], \mathtt{x} = 2, \mathtt{s} = [3]$	$\Rightarrow$	`rev_append [3] (2::[1])`
	$=$	`rev_append [3] [2;1]`
$[3] \neq [], \mathtt{x} = 3, \mathtt{s} = []$	$\Rightarrow$	`rev_append [] (3::[2;1])`
	$=$	`rev_append [] [3;2;1]`
$[] = []$	$\Rightarrow$	`[3;2;1]`

Tri de listes

Tri pas Insertion : principe

Cet algorithme de tri suit de manière naturelle la structure récursive des listes

Soit `l` une liste à trier :

1. si `l` est vide alors elle est déjà triée
2. sinon, `l` est de la forme `x::s` et,
   - on trie récursivement la suite `s` et on obtient une liste triée `s'`
   - on insert `x` au bon endroit dans `s'` et on obtient une liste triée

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste `l` est triée alors `x` est inséré au bon endroit
- On prend pour le moment <= comme relation d'ordre

```
# let rec inserer x l =
   match l with
     [] -> [x]
   | y::s -> if x<=y then x::l else y::(inserer x s);;
```

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste `l` est triée alors `x` est inséré au bon endroit
- On prend pour le moment <= comme relation d'ordre

```
# let rec inserer x l =
   match l with
     [] -> [x]
   | y::s -> if x<=y then x::l else y::(inserer x s);;

val inserer: 'a -> 'a list -> 'a list
```

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste `l` est triée alors `x` est inséré au bon endroit
- On prend pour le moment <= comme relation d'ordre

```
# let rec inserer x l =
   match l with
     [] -> [x]
   | y::s -> if x<=y then x::l else y::(inserer x s);;

val inserer: 'a -> 'a list -> 'a list


# inserer 5 [3;7;10];;
```

Insertion

- La fonction `inserer` permet d'insérer un élément `x` dans une liste `l`
- Si la liste `l` est triée alors `x` est inséré au bon endroit
- On prend pour le moment <= comme relation d'ordre

```
# let rec inserer x l =
   match l with
     [] -> [x]
   | y::s -> if x<=y then x::l else y::(inserer x s);;

val inserer: 'a -> 'a list -> 'a list


# inserer 5 [3;7;10];;

- : int list = [3; 5; 7; 10]
```

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

`inserer 5 [3;7;10]`

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

$$[3;7;10] \neq [], \mathtt{y} = 3, \mathtt{s} = [7;10], 5 > 3 \quad \Rightarrow \quad \begin{array}{l}\texttt{inserer 5 [3;7;10]} \\ \texttt{3::(inserer 5 [7;10])}\end{array}$$

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

		`inserer 5 [3;7;10]`
$[3;7;10] \neq [], \mathtt{y} = 3, \mathtt{s} = [7;10], 5 > 3$	$\Rightarrow$	`3::(inserer 5 [7;10])`
$[7;10] \neq [], \mathtt{y} = 7, \mathtt{s} = [10], 5 <= 7$	$\Rightarrow$	`3::5::[7;10]`

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

$[3;7;10] \neq [], \mathtt{y} = 3, \mathtt{s} = [7;10], 5 > 3$ $\Rightarrow$ `3::(inserer 5 [7;10])`

$[7;10] \neq [], \mathtt{y} = 7, \mathtt{s} = [10], 5 <= 7$ $\Rightarrow$ `3::5::[7;10]`

$\Rightarrow$ `3::[5;7;10]`

`inserer 5 [3;7;10]`

Évaluation de la fonction `inserer`

Évaluation de `inserer 5 [3;7;10]`

$[3;7;10] \neq [], \mathtt{y} = 3, \mathtt{s} = [7;10], 5 > 3$
$[7;10] \neq [], \mathtt{y} = 7, \mathtt{s} = [10], 5 <= 7$

```
                inserer 5 [3;7;10]
⇒   3::(inserer 5 [7;10])
⇒   3::5::[7;10]
⇒   3::[5;7;10]
⇒   [3;5;7;10]
```

Trier une liste

On utilise la fonction `inserer` pour réaliser un tri par insertion d'une liste

```
# let rec trier l =
    match l with
      [] -> []
    | x::s -> inserer x (trier s);;
val trier : 'a list -> 'a list = <fun>
```

Trier une liste

On utilise la fonction `inserer` pour réaliser un tri par insertion d'une liste

```
# let rec trier l =
    match l with
      [] -> []
    | x::s -> inserer x (trier s);;
val trier : 'a list -> 'a list = <fun>

# trier [6; 1; 9; 4; 3];;
- : int list = [1; 3; 4; 6; 9]
```

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

`trier [6;4;1;5]`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

$\texttt{x} = 6, \texttt{s} = [4;1;5]$ $\Rightarrow$

```
trier [6;4;1;5]
inserer 6 (trier [4;1;5])
```

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$\texttt{x} = 6, \texttt{s} = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$\texttt{x} = 4, \texttt{s} = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))))`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`
	$\Rightarrow$	`inserer 6 [1;4;5]`

Évaluation de la fonction `trier`

Évaluation de `trier [6;4;1;5]`

		`trier [6;4;1;5]`
$x = 6, s = [4;1;5]$	$\Rightarrow$	`inserer 6 (trier [4;1;5])`
$x = 4, s = [1;5]$	$\Rightarrow$	`inserer 6 (inserer 4 (trier [1;5]))`
$x = 1, s = [5]$	$\Rightarrow$	`...(inserer 1 (trier [5])))`
$x = 5, s = []$	$\Rightarrow$	`...(inserer 5 (trier [])))`
	$\Rightarrow$	`...(inserer 5 []))`
	$\Rightarrow$	`inserer 6 (inserer 4 (inserer 1 [5]))`
	$\Rightarrow$	`inserer 6 (inserer 4 [1;5])`
	$\Rightarrow$	`inserer 6 [1;4;5]`
	$\Rightarrow$	`[1;4;5;6]`

Tri Rapide : principe

Soit une liste `l` à trier :

1. si `l` est vide alors elle est triée
2. sinon, choisir un élément `p` de la liste (le premier par exemple) nommé le pivot
3. partager `l` en deux listes `g` et `d` contenant les autres éléments de `l` qui sont plus petits (resp. plus grands) que la valeur du pivot `p`
4. trier récursivement `g` et `d`, on obtient deux listes `g'` et `d'` triées
5. on renvoie la liste `g'@[p]@d'` (qui est bien triée)

Partage d'une liste

La fonction suivante permet de partager une liste `l` en deux sous-listes `g` et `d` contenant les éléments de `l` plus petits (resp. plus grands) qu'une valeur donnée `p`

```
#let rec partage p l =
  match l with
     [] -> ([] , [])
    |x::s -> let (g, d) = partage p s in
                if x <= p then (x::g, d) else (g, x::d) ;;
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>
```

Partage d'une liste

La fonction suivante permet de partager une liste `l` en deux sous-listes `g` et `d` contenant les éléments de `l` plus petits (resp. plus grands) qu'une valeur donnée `p`

```
#let rec partage p l =
  match l with
     [] -> ([] , [])
    |x::s -> let (g, d) = partage p s in
                if x <= p then (x::g, d) else (g, x::d) ;;
val partage : 'a -> 'a list -> 'a list * 'a list = <fun>

# partage 5 [1;9;7;3;2;4];;
- : int list * int list = ([1; 3; 2; 4], [9; 7])
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

`partage 5 [1;9;3;7]`

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒   ([3], [7])
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒   ([3], [7])
⇒   let (g2, d2) = ([3],[7]) in (g2, 9::d2)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒   ([3], [7])
⇒   let (g2, d2) = ([3],[7]) in (g2, 9::d2)
⇒   ([3], [9;7])
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒   ([3], [7])
⇒   let (g2, d2) = ([3],[7]) in (g2, 9::d2)
⇒   ([3], [9;7])
⇒   let (g1, d1) = ([3],[9;7]) in (1::g1, d1)
```

Évaluation de la fonction `partage`

Évaluation de `partage 5 [1;9;3;7]`

```
    partage 5 [1;9;3;7]
⇒   let (g1, d1) = partage 5 [9;3;7] in (1::g1, d1)
⇒   let (g2, d2) = partage 5 [3;7] in (g2, 9::d2)
⇒   let (g3, d3) = partage 5 [7] in (3::g3, d3)
⇒   let (g4, d4) = partage 5 [] in (g4, 7::d4)
⇒   [],[]
⇒   let (g4, d4) = ([],[]) in (g4, 7::d4)
⇒   ([], [7])
⇒   let (g3, d3) = ([],[7]) in (3::g3, d3)
⇒   ([3], [7])
⇒   let (g2, d2) = ([3],[7]) in (g2, 9::d2)
⇒   ([3], [9;7])
⇒   let (g1, d1) = ([3],[9;7]) in (1::g1, d1)
⇒   ([1;3], [9;7])
```

Tri rapide

```
# let rec tri_rapide l =
   match l with
      [] -> []
    | p::s -> let g , d = partage p s in
              (tri_rapide g)@[p]@(tri_rapide d)  ;;

val tri_rapide : 'a list -> 'a list = <fun>



# tri_rapide [5; 1; 9; 7; 3; 2; 4];;
- : int list = [1; 2; 3; 4; 5; 7; 9]
```