

## 4 Notions

- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

## 4 Notions

- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

# Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;  
Error: Unbound value fact
```

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

# Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

```
let rec ID P1 ... Pn = EXP
```

où `ID` est un identificateur, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

# Définitions récursives

```
# let fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Error: Unbound value fact

Cette définition pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

Pour pouvoir réaliser des **définitions récursives** (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

```
let rec ID P1 ... Pn = EXP
```

où `ID` est un identificateur, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

```
# let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;  
val fact : int -> int = <fun>
```

Ceci définit bien la fonction factorielle.

```
# (fact 7);;  
- : int = 5040
```

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
```

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
   x)
  + x;;
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
```

```
# let x = 10 in
  (let x = 20 in
   x)
  + x;;
- : int = 30
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  (let x = 20 in
   x)
  + x;;
```

# Définitions récursives

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Comparons les expressions suivantes :

```
# let x = 10 in
  let x = 20 in
    x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let rec x =
  let x = 20 in
    x + x;;
val x : int = 40
```

```
# let x = 10 in
  (let x = 20 in
    x)
  + x;;
- : int = 30
```

```
# let rec x =
  (let x = 20 in
    x)
  + x;;
Error: This kind of expression is
not allowed as right-hand side of
'let rec'
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
Error: Unbound value f
```

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>
```

# Définitions récursives

```
# let f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =  
  let f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
- : int = 9
```

```
# let f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;  
Stack overflow during evaluation  
(looping recursion?).
```

```
# let rec f x =  
  let rec f y =  
    y + (f (x - 1))  
  in  
  if x = 0 then  
    0  
  else  
    x + (f (x - 1));;  
val f : int -> int = <fun>  
# (f 3);;
```

# Définitions récursives

```
# let f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
```

Error: Unbound value f

```
# let rec f x =
  let f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
val f : int -> int = <fun>
# (f 3);;
- : int = 9
```

```
# let f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
```

```
val f : int -> int = <fun>
```

```
# (f 3);;
```

Stack overflow during evaluation  
(looping recursion?).

```
# let rec f x =
  let rec f y =
    y + (f (x - 1))
  in
  if x = 0 then
    0
  else
    x + (f (x - 1));;
```

```
val f : int -> int = <fun>
```

```
# (f 3);;
```

Stack overflow during evaluation  
(looping recursion?).

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions  
mutuellement récursives  
simultanément.

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel (`zero n`) renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `(zero 4)` :

```
(zero 4)
```

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de `(zero 4)` :  
`(zero 4) → (un 3)`

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

`(zero 4) → (un 3) → (deux 2)`

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

`(zero 4) → (un 3) → (deux 2)`  
`→ (zero 1)`

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de

`(zero 4)` :

`(zero 4)` → `(un 3)` → `(deux 2)`  
→ `(zero 1)` → `(un 0)`

# Définitions mutuellement récursives

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

```
# let rec zero x =
  if x = 0 then
    "zero"
  else
    (un (x - 1))
and un x =
  if x = 0 then
    "un"
  else
    (deux (x - 1))
and deux x =
  if x = 0 then
    "deux"
  else
    (zero (x - 1));;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit trois fonctions mutuellement récursives simultanément.

L'appel `(zero n)` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par 3.

Suite d'appels pour le calcul de `(zero 4)` :

`(zero 4)` → `(un 3)` → `(deux 2)`  
→ `(zero 1)` → `(un 0)`  
→ "un".

# Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

# Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

Ici,  $C$  est une condition et  $I$  est une expression.

# Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que C :  
    /  
Fin
```

Ici, *C* est une condition et */* est une expression.

# Simulation des instructions de boucle

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen d'une définition d'une fonction récursive utilisant une conditionnelle et d'un appel à cette fonction :

```
Tant que  $C$  :  
   $I$   
Fin
```

```
Fonction rec  $f$  :  
  Si  $C$  :  
     $I$   
    Appel à  $f$   
  Fin  
Fin  
Appel à  $f$ 
```

Ici,  $C$  est une condition et  $I$  est une expression.

# Simulation des instructions de boucle — exemples

## Boucle `while` simple :

### Fonction C

```
int triangle(int n) {
    int i, res;
    res = 0;
    i = n;
    while (i >= 1) {
        res += i;
        i -= 1;
    }
    return res;
}
```

### Fonction Caml

```
let triangle n =
  let rec aux i =
    if i = 0 then
      0
    else
      i + (aux (i - 1))
  in
  (aux n);;
```

# Simulation des instructions de boucle — exemples

## Boucle `while` simple :

Fonction C

```
int triangle(int n) {
    int i, res;
    res = 0;
    i = n;
    while (i >= 1) {
        res += i;
        i -= 1;
    }
    return res;
}
```

Fonction Caml

```
let triangle n =
  let rec aux i =
    if i = 0 then
      0
    else
      i + (aux (i - 1))
  in
  (aux n);;
```

## Boucle `for` simple :

Fonction C

```
int somme_paires(int n) {
    int i, res;
    res = 0;
    for (i = 0; i <= n; i += 2) {
        res += i;
    }
    return res;
}
```

Fonction Caml

```
let somme_paires n =
  let rec aux i =
    if i > n then
      0
    else
      i + (aux (i + 2))
  in
  (aux 0);;
```

# Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

# Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

# Récurivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

```
(fact 4)
```

# Récurivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

`(fact 4)` → `4 * (fact 3)`

# Récurivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

$(\text{fact } 4) \rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2)$

# Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1)\end{aligned}$$

# Récurtivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1\end{aligned}$$

# Récurivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et  
l'appel

```
(fact 4);;
```

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1 \\ &\rightsquigarrow 24\end{aligned}$$

# Récurivité terminale — motivation

```
let rec fact n =  
  if n <= 1 then  
    1  
  else  
    n * (fact (n - 1));;
```

Considérons la fonction ci-contre et l'appel

```
(fact 4);;
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4) &\rightarrow 4 * (\text{fact } 3) \rightarrow 4 * 3 * (\text{fact } 2) \\ &\rightarrow 4 * 3 * 2 * (\text{fact } 1) \rightarrow 4 * 3 * 2 * 1 \\ &\rightsquigarrow 24\end{aligned}$$

Ce calcul, pour être mené à bien, a dû **garder en mémoire l'expression**

$$4 * 3 * 2 * 1$$

qui fait intervenir quatre (=  $n$ ) opérandes et trois (=  $n - 1$ ) opérateurs.

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

(fact 4 1)

# Récurivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

`(fact 4 1)` → `(fact 3 (4 * 1))`

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

$$(\text{fact } 4 \ 1) \rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4)$$

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4 \ 1) &\rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4) \\ &\rightarrow (\text{fact } 2 \ (3 * 4))\end{aligned}$$

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4 \ 1) &\rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4) \\ &\rightarrow (\text{fact } 2 \ (3 * 4)) \rightsquigarrow (\text{fact } 2 \ 12)\end{aligned}$$

# Récurtivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4 \ 1) &\rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4) \\ &\rightarrow (\text{fact } 2 \ (3 * 4)) \rightsquigarrow (\text{fact } 2 \ 12) \\ &\rightarrow (\text{fact } 1 \ (2 * 12))\end{aligned}$$

# Récurivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récurifs en

$$\begin{aligned}(\text{fact } 4 \ 1) &\rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4) \\ &\rightarrow (\text{fact } 2 \ (3 * 4)) \rightsquigarrow (\text{fact } 2 \ 12) \\ &\rightarrow (\text{fact } 1 \ (2 * 12)) \rightsquigarrow (\text{fact } 1 \ 24)\end{aligned}$$

# Récurivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et  
l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

```
(fact 4 1)  →  (fact 3 (4 * 1))  ⇝  (fact 3 4)  
            →  (fact 2 (3 * 4))  ⇝  (fact 2 12)  
            →  (fact 1 (2 * 12)) ⇝  (fact 1 24)  
            →  24
```

# Récurivité terminale — motivation

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));; (fact 4 1);;
```

Considérons la fonction ci-contre et l'appel

Cette dernière expression s'évalue au fil des appels récursifs en

$$\begin{aligned}(\text{fact } 4 \ 1) &\rightarrow (\text{fact } 3 \ (4 * 1)) \rightsquigarrow (\text{fact } 3 \ 4) \\ &\rightarrow (\text{fact } 2 \ (3 * 4)) \rightsquigarrow (\text{fact } 2 \ 12) \\ &\rightarrow (\text{fact } 1 \ (2 * 12)) \rightsquigarrow (\text{fact } 1 \ 24) \\ &\rightarrow 24\end{aligned}$$

Ce calcul, pour être mené à bien, a dû **garder en mémoire des expressions** faisant intervenir au plus deux opérandes et un opérateur, en plus de l'appel de fonction.

# Réversivité terminale

La 2<sup>e</sup> version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

# Réversivité terminale

La 2<sup>e</sup> version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1<sup>re</sup> version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

# Récurtivité terminale

La 2<sup>e</sup> version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1<sup>re</sup> version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2<sup>e</sup>, elle ne subit aucune modification

```
(fact (n - 1) (n * acc)).
```

# Récurtivité terminale

La 2<sup>e</sup> version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1<sup>re</sup> version, la valeur de retour subit une multiplication

$$n * (\text{fact } (n - 1))$$

alors que dans la 2<sup>e</sup>, elle ne subit aucune modification

$$(\text{fact } (n - 1) (n * \text{acc})).$$

Le calcul de la 1<sup>re</sup> version nécessite de garder en mémoire une expression de taille  $\Theta(n)$ , alors que celui de la 2<sup>e</sup> ne travaille que sur une expression de taille  $\Theta(1)$ .

# Récurtivité terminale

La 2<sup>e</sup> version de la fonction `fact` possède la propriété d'être **réursive terminale** : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1<sup>re</sup> version, la valeur de retour subit une multiplication

$$n * (\text{fact } (n - 1))$$

alors que dans la 2<sup>e</sup>, elle ne subit aucune modification

$$(\text{fact } (n - 1) (n * \text{acc})).$$

Le calcul de la 1<sup>re</sup> version nécessite de garder en mémoire une expression de taille  $\Theta(n)$ , alors que celui de la 2<sup>e</sup> ne travaille que sur une expression de taille  $\Theta(1)$ .

Les fonctions récursives terminales utilisent **moins de mémoire** que leurs analogues non récursives terminales. Elles sont donc à préférer.

## Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

## Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels.

C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur.

## Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur.

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

# Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur.

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));;
```

# Récurtivité terminale — accumulateurs

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'**accumulateur**.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur.

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable :

la fonction

```
let rec fact n acc =  
  if n <= 1 then  
    acc  
  else  
    (fact (n - 1) (n * acc));;
```

devient

```
let fact n =  
  let rec aux n acc =  
    if n <= 1 then  
      acc  
    else  
      (aux (n - 1) (n * acc))  
  in  
  (aux n 1);;
```

# Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2));;
```

C'est la version non récursive terminale de la fonction calculant le  $n^e$  nombre de Fibonacci.

# Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2));;
```

C'est la version non récursive terminale de la fonction calculant le  $n^e$  nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

# Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2));;
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0);;
```

C'est la version non récursive terminale de la fonction calculant le  $n^e$  nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version récursive terminale de la fonction précédente.

# Récurtivité terminale — Fibonacci

```
let rec fibo n =  
  if n <= 1 then  
    n  
  else  
    (fibo (n - 1))  
    + (fibo (n - 2));;
```

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
        (acc1 + acc2) acc1)  
  in  
  (aux n 1 0);;
```

C'est la version non réursive terminale de la fonction calculant le  $n^e$  nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

C'est la version réursive terminale de la fonction précédente.

Elle utilise deux accumulateurs (à cause du double appel récursif précédent). `acc1` contient la valeur du  $n - 1^e$  nombre de Fibonacci et `acc2` contient la valeur du  $n - 2^e$  nombre de Fibonacci.

# Réversivité terminale — forme générale

Une fonction récursive terminale a pour **forme générale**

```
Fonction rec  $f(x_1, \dots, x_n, acc_1, \dots, acc_m)$  :  
  Si  $C$  :  
     $f(\text{maje}(x_1, \dots, x_n), \text{maj}_s(acc_1, \dots, acc_m))$   
  Sinon :  
     $R$   
Fin  
Fin
```

où

- $x_1, \dots, x_n$  sont les paramètres (entrées);
- $acc_1, \dots, acc_m$  sont les accumulateurs (sorties);
- $R$  est une expression résultat obtenue à partir des accumulateurs;
- $\text{maje}$  indique la mise à jour des  $x_1, \dots, x_n$  lors de l'appel récursif;
- $\text{maj}_s$  indique la mise à jour des  $acc_1, \dots, acc_m$  lors de l'appel récursif.

# Réversivité terminale — dérécursivatiion

La **dérécursivatiion** est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Les notations sont ici les mêmes que celles utilisées précédemment.

# Réversivité terminale — dérécursivisation

La **dérécursivisation** est un procédé qui permet de transformer toute fonction réursive terminale en une fonction itérative.

Voici une fonction réursive terminale dans sa forme générale et sa version dérécursivée :

```
Fonction rec  $f(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m)$  :  
  Si  $C$  :  
     $f(\text{maje}(x_1, \dots, x_n),$   
       $\text{majs}(acc_1, \dots, acc_m))$   
  Sinon :  
     $R$   
Fin  
Fin
```

```
Fonction it  $g(x_1, \dots, x_n,$   
           $acc_1, \dots, acc_m)$  :  
  Tant que  $C$  :  
     $(x_1, \dots, x_n)$   
       $:= \text{maje}(x_1, \dots, x_n)$   
     $(acc_1, \dots, acc_m)$   
       $:= \text{majs}(acc_1, \dots, acc_m)$   
  Fin  
   $R$   
Fin
```

Les notations sont ici les mêmes que celles utilisées précédemment.

# Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
       (acc1 + acc2) acc1)  
  in  
  (aux n 1 0);;
```

# Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
       (acc1 + acc2) acc1)  
  in  
  (aux n 1 0);;
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    (fibo (n - 1)  
     (acc1 + acc2) acc1)  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1;;
```

# Récurtivité terminale — dérécursivation (exemple)

Fonction en forme habituelle :

```
let fibo n =  
  let rec aux n acc1 acc2 =  
    if n = 0 then  
      acc2  
    else if n = 1 then  
      acc1  
    else  
      (aux (n - 1)  
       (acc1 + acc2) acc1)  
  in  
  (aux n 1 0);;
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =  
  if n >= 2 then  
    (fibo (n - 1)  
     (acc1 + acc2) acc1)  
  else  
    if n = 0 then  
      acc2  
    else  
      acc1;;
```

Version dérécursivée en C :

```
int fibo(int n, int acc1, int acc2) {  
  while (n >= 2) {  
    n = n - 1;  
    acc1 = acc1 + acc2;  
    acc2 = acc1 - acc2;  
  }  
  if (n == 0) return acc2;  
  else return acc1;  
}
```

## 4 Notions

- Récursivité
- Filtrage
- Fonctions d'ordre supérieur
- Polymorphisme
- Stratégies d'évaluation

# Exemple introductif

Considérons le type

```
type point =  
  |Seg of int  
  |Plan of int * int  
  |Espace of int * int * int;;
```

# Exemple introductif

Considérons le type

```
type point =  
  |Seg of int  
  |Plan of int * int  
  |Espace of int * int * int;;
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

# Exemple introductif

Considérons le type

```
type point =  
  |Seg of int  
  |Plan of int * int  
  |Espace of int * int * int;;
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Seg x) -> (Seg (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z));;
```

# Exemple introductif

Considérons le type

```
type point =  
  |Seg of int  
  |Plan of int * int  
  |Espace of int * int * int;;
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Seg x) -> (Seg (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z));;
```

```
# (oppose (Plan (3,1)));;  
- : point = Plan (-3, -1)
```

# Exemple introductif

Considérons le type

```
type point =  
  |Seg of int  
  |Plan of int * int  
  |Espace of int * int * int;;
```

On souhaite écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

La meilleure manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =  
  match p with  
  |(Seg x) -> (Seg (-x))  
  |(Plan (x, y)) -> (Plan (-x, -y))  
  |(Espace (x, y, z)) -> (Espace (-x, -y, -z));;  
  
# (oppose (Plan (3,1)));;          # (oppose (Espace(1, 0, -1)));;  
- : point = Plan (-3, -1)        - : point = Espace (-1, 0, 1)
```

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with  
  |MOTIF1 -> EXP1  
  ...  
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

- 1 `EXP` est évaluée ;

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

- 1 `EXP` est évaluée ;
- 2 on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas ;

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

- 1 `EXP` est évaluée ;
- 2 on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas ;
- 3 si un motif `MOTIFi` filtre la valeur de `EXP`, la valeur de toute l'expression est celle de `EXPi` ;

# Syntaxe et évaluation

La construction syntaxique

```
match EXP with
  |MOTIF1 -> EXP1
  ...
  |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un **filtrage de motifs** sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une **clause**.

L'évaluation de cette expression se déroule de la manière suivante :

- 1 `EXP` est évaluée ;
- 2 on essaye de **filtrer** (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas ;
- 3 si un motif `MOTIFi` filtre la valeur de `EXP`, la valeur de toute l'expression est celle de `EXPi` ;
- 4 si aucun motif ne filtre la valeur de `EXP`, une erreur est signalée (à l'exécution).

# Principe du filtrage

Le filtrage de motifs peut se penser en 1<sup>re</sup> approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

# Principe du filtrage

Le filtrage de motifs peut se penser en 1<sup>re</sup> approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

- 1 lorsque le traitement à effectuer dépend davantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

# Principe du filtrage

Le filtrage de motifs peut se penser en 1<sup>re</sup> approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

- 1 lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =                               renvoie le nombre de
  match p with                                   coordonnées du point p.
  | (Seg x) -> 1
  | (Plan (x, y)) -> 2
  | (Espace (x, y, z)) -> 3;;
```

# Principe du filtrage

Le filtrage de motifs peut se penser en 1<sup>re</sup> approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

- 1 lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =                               renvoie le nombre de
  match p with                                   coordonnées du point p.
  | (Seg x) -> 1
  | (Plan (x, y)) -> 2
  | (Espace (x, y, z)) -> 3;;
```

- 2 lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

# Principe du filtrage

Le filtrage de motifs peut se penser en 1<sup>re</sup> approximation comme le `switch` du C. Il est dans les faits beaucoup plus puissant.

On l'utilise principalement :

- 1 lorsque le traitement à effectuer dépend d'avantage de la **structure d'une valeur** que de la valeur elle-même. P.ex.,

```
let dimension p =                               renvoie le nombre de
  match p with                                   coordonnées du point p.
  | (Seg x) -> 1
  | (Plan (x, y)) -> 2
  | (Espace (x, y, z)) -> 3;;
```

- 2 lorsque l'on souhaite d'accéder à une **partie d'une valeur**, le filtrage permettant de **déconstruire**. P.ex.,

```
let premiere_coordonnee p =                     renvoie la première coordonnée
  match p with                                   du point p.
  | (Seg x) -> x
  | (Plan (x, y)) -> x
  | (Espace (x, y, z)) -> x;;
```

# Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif** .

# Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif** .

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =  
  match p with  
    | (Seg x) -> 1  
    | (Espace (x, y, z)) -> 3;;  
Warning 8: this pattern-matching is not exhaustive.  
Here is an example of a value that is not matched:  
Plan (_, _)  
val dimension : point -> int = <fun>
```

# Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif** .

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =
  match p with
  | (Seg x) -> 1
  | (Espace (x, y, z)) -> 3;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Plan (_, _)
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

# Exhaustivité du filtrage

L'interpréteur vérifie si le filtrage est **exhaustif**, c.-à-d. si toute expression du type attendu **peut être filtrée par au moins un motif** .

Si ça n'est pas le cas, un avertissement est signalé. P.ex.,

```
# let dimension p =
  match p with
    | (Seg x) -> 1
    | (Espace (x, y, z)) -> 3;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Plan (_, _)
val dimension : point -> int = <fun>
```

Le **joker** `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs. P.ex.,

```
# let entier_vers_chaine n =
  match n with
    | 0 -> "zero"
    | 1 -> "un"
    | 2 -> "deux"
    | _ -> "autre";;
val entier_vers_chaine : int -> string = <fun>
```

Il est possible de **raffiner le filtrage** en incluant des tests à une clause

MOTIF -> EXP.

Il est possible de **raffiner le filtrage** en incluant des tests à une clause

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Il est possible de **raffiner le filtrage** en incluant des tests à une clause

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de TEST soit `true`.

Il est possible de **raffiner le filtrage** en incluant des tests à une clause

MOTIF -> EXP.

On utilise pour cela la syntaxe

MOTIF when TEST -> EXP

où TEST est une expression de type `bool` appelée **garde**.

Pour que ce motif filtre une expression, il faut en plus que la valeur de TEST soit `true`.

P.ex.,

```
let est_dans_quart_de_plan p =  
  match p with  
  | (Seg _) -> false  
  | (Plan (x, y)) when x >= 0 && y >= 0 -> true  
  | (Plan (_, _)) -> false  
  | (Espace (_, _, _)) -> false;;
```

teste si l'argument est un point du plan à coordonnées positives.

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (0, true, (), 'a', "abc", etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement **x**, **y** et/ou **z** à des valeurs.

```
let somme p =  
  match p with  
  | (Seg x) -> x  
  | (Plan (x, y)) -> x + y  
  | (Espace (x, y, z)) ->  
    x + y + z;;
```

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |(Seg x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z;;
```

Attention : ici  
l'occurrence de `n` dans le  
2<sup>e</sup> motif ne fait pas  
référence à la liaison  
précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1;;
```

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, *etc.*);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |(Seg x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z;;
```

Attention : ici  
l'occurrence de `n` dans le  
2<sup>e</sup> motif ne fait pas  
référence à la liaison  
précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1;;  
# (f 0);;  
- : int = 0
```

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |(Seg x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z;;
```

Attention : ici  
l'occurrence de `n` dans le  
2<sup>e</sup> motif ne fait pas  
référence à la liaison  
précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1;;  
# (f 0);;  
- : int = 0  
# (f 3);;  
- : int = -1
```

# Plus de précisions sur les motifs

Il existe trois sortes de motifs :

- 1 les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, etc.);
- 2 les motifs à **paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers;
- 3 les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

```
let somme p =  
  match p with  
  |(Seg x) -> x  
  |(Plan (x, y)) -> x + y  
  |(Espace (x, y, z)) ->  
    x + y + z;;
```

Attention : ici l'occurrence de `n` dans le 2<sup>e</sup> motif ne fait pas référence à la liaison précédente.

```
let f x =  
  let n = 2 in  
  match x with  
  |0 -> 0  
  |n -> -1  
  |_ -> 1;;
```

```
# (f 0);;  
- : int = 0  
# (f 3);;  
- : int = -1
```

Ainsi, le motif `n` filtre toutes les valeurs.

## Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

## Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une formule est une donnée récursive :

- 1 c'est un atome  $P$  ;
- 2 ou bien est la négation d'une formule ( $\neg F$ ) ;
- 3 ou bien est la conjonction de deux formules ( $F \wedge G$ ) ;
- 4 ou bien est la disjonction de deux formules ( $F \vee G$ ).

# Exemple : évaluation de formules

On souhaite représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une formule est une donnée récursive :

- 1 c'est un atome  $P$  ;
- 2 ou bien est la négation d'une formule ( $\neg F$ ) ;
- 3 ou bien est la conjonction de deux formules ( $F \wedge G$ ) ;
- 4 ou bien est la disjonction de deux formules ( $F \vee G$ ).

On en déduit la définition de type (somme à paramètres et récursive) suivante :

```
type formule =  
  |Atome of char  
  |Non of formule  
  |Et of formule * formule  
  |Ou of formule * formule;;
```

## Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

## Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool;;
```

## Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool;;
```

La fonction d'**évaluation** d'une formule sous une valuation s'écrit très simplement au moyen d'un filtrage :

## Exemple : évaluation de formules

On code une **valuation** par une fonction qui associe à un caractère (atome) sa valeur de vérité :

```
type valuation = char -> bool;;
```

La fonction d'**évaluation** d'une formule sous une valuation s'écrit très simplement au moyen d'un filtrage :

```
let rec evaluer form valu =  
  match form with  
  | (Atome c) -> (valu c)  
  | (Non f) -> (not (evaluer f valu))  
  | (Et (f, g)) -> (evaluer f valu) && (evaluer g valu)  
  | (Ou (f, g)) -> (evaluer f valu) || (evaluer g valu);;
```

## Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation  $v$

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

## Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation  $v$

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela,  $f$  est codée par

let  $f =$

```
(Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R'))));;
```

# Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation  $v$

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela,  $f$  est codée par

```
let f =  
  (Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R'))));;
```

et  $v$  par

```
let v c =  
  match c with  
  | 'P' -> false  
  | 'R' -> true  
  | _ -> false;;
```

# Exemple : évaluation de formules

On peut l'utiliser sur la formule

$$f := (\neg P) \wedge (P \vee R)$$

et la valuation  $v$

$$P \mapsto \text{faux}, \quad R \mapsto \text{vrai}$$

Pour cela,  $f$  est codée par

```
let f =  
  (Et ((Non (Atome 'P')), (Ou (Atome 'P', Atome 'R'))));;
```

et  $v$  par

```
let v c =  
  match c with  
  | 'P' -> false  
  | 'R' -> true  
  | _ -> false;;
```

L'évaluation de  $f$  sous  $v$  donne

```
# (evaluer f v);;  
- : bool = true
```

- 4 Notions
  - Récursivité
  - Filtrage
  - Fonctions d'ordre supérieur
  - Polymorphisme
  - Stratégies d'évaluation

# Définition

Une **fonction d'ordre supérieur** est une fonction  $f$  qui vérifie au moins l'une des deux conditions suivantes :

- 1  $f$  possède un paramètre de type fonction ;
- 2  $f$  renvoie une fonction.

# Définition

Une **fonction d'ordre supérieur** est une fonction  $f$  qui vérifie au moins l'une des deux conditions suivantes :

- 1  $f$  possède un paramètre de type fonction ;
- 2  $f$  renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

# Définition

Une **fonction d'ordre supérieur** est une fonction  $f$  qui vérifie au moins l'une des deux conditions suivantes :

- 1  $f$  possède un paramètre de type fonction ;
- 2  $f$  renvoie une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement générique.

Le fait de pouvoir renvoyer une fonction est un procédé très puissant en programmation fonctionnelle. Le programmeur n'est plus le seul concepteur de fonctions : l'exécution/l'interprétation peut en créer à la volée et en appeler.

# Fonctions curryfiées

Rappelons que si  $f$  est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec  $1 \leq k \leq n - 1$  produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

# Fonctions curryfiées

Rappelons que si  $f$  est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec  $1 \leq k \leq n - 1$  produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont **curryfiées** en Caml : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions**.

# Fonctions curryfiées

Rappelons que si  $f$  est une fonction de type

$$E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n \rightarrow S,$$

toute application partielle

$$(f \ e_1 \ e_2 \ \dots \ e_k)$$

avec  $1 \leq k \leq n - 1$  produit une valeur de type

$$E_{k+1} \rightarrow \dots \rightarrow E_n \rightarrow S$$

qui est donc une fonction.

Les fonctions à plusieurs paramètres sont **curryfiées** en Caml : elles se comportent comme des fonctions à un seul paramètre qui **renvoient des fonctions**.

On peut donc voir toute fonction à deux paramètres ou plus comme une fonction d'ordre supérieur car son application partielle renvoie une fonction.

# Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    (f (appli_repetee f x (n - 1))));;
```

# Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    (f (appli_repetee f x (n - 1))));;
```

On infère le type

`(int -> int) -> int -> int -> int,`

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

# Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    (f (appli_repetee f x (n - 1)));;
```

On infère le type

`(int -> int) -> int -> int -> int,`

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée  $n^e$  de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(f \circ \dots \circ f)}_{n \text{ fois}}(x)$$

# Fonctions paramétrées par des fonctions

Analysons le type de la fonction

```
let rec appli_repetee f x n =  
  if n = 0 then  
    x + 0  
  else  
    (f (appli_repetee f x (n - 1)));;
```

On infère le type

`(int -> int) -> int -> int -> int,`

ce qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée  $n^e$  de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(f \circ \dots \circ f)}_{n \text{ fois}}(x)$$

```
# (appli_repetee (fun x -> 2 * x) 3 4);;  
- : int = 48
```

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w);;
```

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w);;
```

On infère le type

```
string -> string -> string -> string.
```

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w);;
```

On infère le type

`string -> string -> string -> string.`

Il est équivalent (à cause du parenthésage implicite) au type

`string -> string -> (string -> string),`

Ce qui montre que `encadrer` renvoie une fonction `string -> string`.  
C'est donc une fonction d'ordre supérieur.

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
  (fun v -> u ^ v ^ w);;
```

On infère le type

```
string -> string -> string -> string.
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string),
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`.  
C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
    (fun v -> u ^ v ^ w);;
```

On infère le type

```
string -> string -> string -> string.
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string),
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`.  
C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = (encadrer "aa" "bb");;  
val f : string -> string = <fun>
```

# Fonctions renvoyant des fonctions

Analysons le type de la fonction

```
let encadrer u w =  
    (fun v -> u ^ v ^ w);;
```

On infère le type

```
string -> string -> string -> string.
```

Il est équivalent (à cause du parenthésage implicite) au type

```
string -> string -> (string -> string),
```

Ce qui montre que `encadrer` renvoie une fonction `string -> string`.  
C'est donc une fonction d'ordre supérieur.

L'appel `(encadrer u v)` renvoie une fonction acceptant une chaîne de caractères et renvoyant la chaîne de caractères encadrée par `u` et `v`.

```
# let f = (encadrer "aa" "bb");;  
val f : string -> string = <fun>  
# (f "bab");;  
- : string = "aababbb"
```

## Exemple complet : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable  $t$  de degré possiblement infini et à coefficients entiers.

## Exemple complet : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable  $t$  de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les  $\alpha_j$  sont des entiers.

## Exemple complet : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable  $t$  de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les  $\alpha_j$  sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

# Exemple complet : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable  $t$  de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les  $\alpha_j$  sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geq 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \dots$$

## Exemple complet : séries génératrices

On souhaite représenter des **séries génératrices**. Ce sont des polynômes en une variable  $t$  de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geq 0} \alpha_n t^n$$

où les  $\alpha_j$  sont des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de coder des suites d'entiers

$$(\alpha_0, \alpha_1, \alpha_2, \dots).$$

P.ex., la série génératrice de la suite des puissances de 2 est

$$\sum_{n \geq 0} 2^n t^n = 1 + 2t + 4t^2 + 8t^3 + 16t^4 + \dots$$

**Question** : comment représenter des séries génératrices ?

## Exemple complet : séries génératrices

**Réponse** : par une **fonction** qui à tout entier positif  $n$  associe le coefficient  $\alpha_n$  de  $t^n$ .

## Exemple complet : séries génératrices

**Réponse** : par une **fonction** qui à tout entier positif  $n$  associe le coefficient  $\alpha_n$  de  $t^n$ .

C'est le type

```
type serie_g = int -> int;;
```

## Exemple complet : séries génératrices

**Réponse** : par une **fonction** qui à tout entier positif  $n$  associe le coefficient  $\alpha_n$  de  $t^n$ .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

## Exemple complet : séries génératrices

**Réponse** : par une **fonction** qui à tout entier positif  $n$  associe le coefficient  $\alpha_n$  de  $t^n$ .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

## Exemple complet : séries génératrices

**Réponse** : par une **fonction** qui à tout entier positif  $n$  associe le coefficient  $\alpha_n$  de  $t^n$ .

C'est le type

```
type serie_g = int -> int;;
```

En effet, pour connaître une série génératrice, il suffit de connaître chacun de ses coefficients.

Il n'est pas possible de les représenter dans une liste à cause du caractère infini de ces objets (degré possiblement infini).

Par exemple, la série génératrice des puissances de 2 est ainsi codée par

```
let puissances_2 =  
  (fun n -> (int_of_float (2. ** (float_of_int n))));;
```

## Exemple complet : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

# Exemple complet : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

**1** somme :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) + \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

# Exemple complet : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

**1** somme :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) + \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

**2** produit d'Hadamard :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) \cdot \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

# Exemple complet : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

**1** somme :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) + \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

**2** produit d'Hadamard :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) \cdot \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

**3** produit :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) \cdot \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \sum_{i=0}^k \alpha_i \beta_{k-i} t^k .$$

# Exemple complet : séries génératrices

Il existe plusieurs **opérations** sur les séries génératrices :

**1** somme :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) + \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} (\alpha_k + \beta_k) t^k ;$$

**2** produit d'Hadamard :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) \cdot \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \alpha_k \beta_k t^k ;$$

**3** produit :

$$\left( \sum_{n \geq 0} \alpha_n t^n \right) \cdot \left( \sum_{m \geq 0} \beta_m t^m \right) = \sum_{k \geq 0} \sum_{i=0}^k \alpha_i \beta_{k-i} t^k .$$

Il est possible de les implanter simplement en utilisant les fonctions d'ordre supérieur.

## Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k));;
```

## Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res;;
```

## Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res;;  
  
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>
```

## Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res;;  
  
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>  
# (s3 3);;  
- : int = 16
```

## Exemple complet : séries génératrices

On implante la somme au moyen d'une fonction anonyme :

```
let somme s1 s2 =  
  (fun k -> (s1 k) + (s2 k));;
```

ou bien sans (avec exactement le même comportement) :

```
let somme s1 s2 =  
  let res k =  
    (s1 k) + (s2 k)  
  in  
  res;;
```

```
# let s3 = (somme puissances_2 puissances_2);;  
val s3 : int -> int = <fun>  
# (s3 3);;  
- : int = 16
```

L'implantation du produit d'Hadamard utilise les mêmes idées :

```
let produit_hadamard s1 s2 =  
  (fun k -> (s1 k) * (s2 k));;
```

## Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    (aux 0)
  in
  resultat;;
```

## Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    (aux 0)
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>
```

## Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    (aux 0)
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>
```

# Exemple complet : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction récursive.

```
let produit s1 s2 =
  let resultat k =
    let rec aux i =
      if i > k then
        0
      else
        (aux (i + 1)) + (s1 i) * (s2 (k - i))
    in
    (aux 0)
  in
  resultat;;

# let sg_un = (fun n -> 1);;
val sg_un : 'a -> int = <fun>

# let sg_un_carre = (produit sg_un sg_un);;
val sg_un_carre : int -> int = <fun>

# (sg_un_carre 0), (sg_un_carre 1), (sg_un_carre 2), (sg_un_carre 3);;
- : int * int * int * int = (1, 2, 3, 4)
```