Plan

Exemple introductif

Considérons le type

```
type point =
    |Droite of int
    |Plan of int * int
    |Espace of int * int * int
```

Exemple introductif

Considérons le type

```
type point =
    |Droite 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 =
    |Droite 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
        |(Droite x) -> (Droite (-x))
        |(Plan (x, y)) -> (Plan (-x, -y))
        |(Espace (x, y, z)) -> (Espace (-x, -y, -z))
```

Exemple introductif

Considérons le type

```
type point =
    |Droite 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
        |(Droite x) -> (Droite (-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 =
    |Droite 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
        |(Droite x) -> (Droite (-x))
        |(Plan (x, y)) -> (Plan (-x, -y))
        |(Espace (x, y, z)) -> (Espace (-x, -y, -z))
```

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

```
# (oppose (Espace(1, 0, -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 1re 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 1re 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.,

Principe du filtrage

Le filtrage de motifs peut se penser en 1re 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 =
    match p with
        |(Droite x) -> 1
        |(Plan (x, y)) -> 2
        |(Espace (x, y, z)) -> 3
```

renvoie le nombre de coordonnées du point `p`.

Principe du filtrage

Le filtrage de motifs peut se penser en 1re 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 =
    match p with
        |(Droite x) -> 1
        |(Plan (x, y)) -> 2
        |(Espace (x, y, z)) -> 3
```

renvoie le nombre de coordonnées du point `p`.

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 1re 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 =
    match p with
        |(Droite x) -> 1
        |(Plan (x, y)) -> 2
        |(Espace (x, y, z)) -> 3
```

renvoie le nombre de coordonnées du point `p`.

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

```
let projection_x p =
    match p with
        |(Droite x) -> x
        |(Plan (x, y)) -> x
        |(Espace (x, y, z)) -> x
```

renvoie la première coordonnée du point `p`.

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
          |(Droite 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
          |(Droite 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
          |(Droite 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>
```

Gardes

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

```
MOTIF -> EXP.
```

Gardes

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

```
MOTIF -> EXP.
```

On utilise pour cela la syntaxe

```
MOTIF when TEST -> EXP
```

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

Gardes

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

```
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`.

Gardes

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

`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
        |(Droite _) -> 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
        |(Droite 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
        |(Droite x) -> x
        |(Plan (x, y)) -> x + y
        |(Espace (x, y, z)) ->
            x + y + z
```

Attention : ici l'occurrence de `n` dans le 2e 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
        |(Droite x) -> x
        |(Plan (x, y)) -> x + y
        |(Espace (x, y, z)) ->
            x + y + z
```

Attention : ici l'occurrence de `n` dans le 2e 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
        |(Droite x) -> x
        |(Plan (x, y)) -> x + y
        |(Espace (x, y, z)) ->
            x + y + z
```

Attention : ici l'occurrence de `n` dans le 2e 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
        |(Droite x) -> x
        |(Plan (x, y)) -> x + y
        |(Espace (x, y, z)) ->
            x + y + z
```

Attention : ici l'occurrence de `n` dans le 2e 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
```

Plan

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

```
E1 -> E2 -> ... -> En -> S,
```

toute application partielle

```
(f e1 e2 ... ek)
```

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

```
Ek+1 -> ... -> En -> S
```

qui est donc une fonction.

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

```
E1 -> E2 -> ... -> En -> S,
```

toute application partielle

```
(f e1 e2 ... ek)
```

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

```
Ek+1 -> ... -> En -> 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** (donc de type `E1 -> (E2 -> ... -> En -> S)`).

Fonctions curryfiées

Rappelons que si `f` est une fonction de type

```
E1 -> E2 -> ... -> En -> S,
```

toute application partielle

```
(f e1 e2 ... ek)
```

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

```
Ek+1 -> ... -> En -> 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** (donc de type `E1 -> (E2 -> ... -> En -> S)`).

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 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"
```

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

$$\texttt{(int -> int) -> int -> int -> int},$$

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 $\texttt{n}^{\text{e}}$ de `f` sur l'entier `x`, c.-à-d.,

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

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

$$\texttt{(int -> int) -> int -> int -> int},$$

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 $\texttt{n}^{\texttt{e}}$ de `f` sur l'entier `x`, c.-à-d.,

$$\underbrace{(\texttt{f} \circ \cdots \circ \texttt{f})}_{\texttt{n fois}}(\texttt{x})$$

```
# (appli_repetee (fun x -> x + 1) 3 4);;
- : int = 7
# (appli_repetee (fun x -> 2 * x) 3 4);;
- : int = 48
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

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

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

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

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\mathtt{f}(\mathtt{g}(\mathtt{f}(\mathtt{g}(x)))).$$

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

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

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\mathtt{f}(\mathtt{g}(\mathtt{f}(\mathtt{g}(x)))).$$

```
# let h = (entrelacer (fun x -> x * 2) (fun x -> x + 1));;
val h : int -> int = <fun>
```

Fonctions paramétrées par et renvoyant des fonctions

Analysons le type de la fonction

```
let entrelacer f g =
    fun x -> 0 + f (0 + (g (f (g x))))
```

On infère le type

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

ce qui montre que `entrelacer` est paramétrée par deux fonctions `int -> int` et renvoie une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

L'appel `(entrelacer f g)` renvoie une fonction qui accepte un entier `x` et renvoie

$$\mathtt{f}(\mathtt{g}(\mathtt{f}(\mathtt{g}(x)))).$$

```
# let h = (entrelacer (fun x -> x * 2) (fun x -> x + 1));;
val h : int -> int = <fun>
# (h 3);;
- : int = 18
```

Exemple complet 1 : mots fonctionnels

Reprenons la façon fonctionnelle de représenter les mots vue précédemment :

```
type 'a mot = {
    lettres : int -> 'a;
    longueur : int
}
```

On rappelle que si `u` est un mot, l'expression `u.lettres i` a pour valeur la `i`$^{\text{e}}$ lettre de `u`.

Exemple complet 1 : mots fonctionnels

Reprenons la façon fonctionnelle de représenter les mots vue précédemment :

```
type 'a mot = {
    lettres : int -> 'a;
    longueur : int
}
```

On rappelle que si `u` est un mot, l'expression `u.lettres i` a pour valeur la i^e lettre de `u`.

Par exemple,

```
let mot_3 = {
    lettres =
        (fun i ->
            match i with
                |1 -> 'a'
                |2 -> 'b'
                |_ -> 'b'
        );
    longueur = 3
}
```

représente le mot (de lettres de type `char`) `abb`.

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string`;

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string`;
- `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char`;

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string`;
- `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char`;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string`;
- `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char`;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

P.ex.,

```
# string_of_mot mot_3 (fun x -> x);;
- : string = "abb"
```

Exemple complet 1 : mots fonctionnels

Voici une fonction qui convertit un mot en une chaîne de caractères qui le représente :

```
let string_of_mot u lettre_vers_char =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = (String.make 1 (lettre_vers_char (u.lettres i))) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- Cette fonction est de type `'a mot -> ('a -> char) -> string`;
- `lettre_vers_char` est une fonction qui convertit une valeur de type `'a` en un `char`;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` constituée de caractères `c`.

P.ex.,

```
# string_of_mot mot_3 (fun x -> x);;
- : string = "abb"

# string_of_mot {lettres = (fun i -> true); longueur = 4}
    (fun b -> if b then "T" else "F");;
- : string = "TTT"
```

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot`;

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

Exemple complet 1 : mots fonctionnels

Voici à présent une fonction de concaténation de mots. Elle renvoie le mot obtenu en concaténant les mots en arguments :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- son type est `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`, se basant sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

P.ex.,

```
# let mot_4 = concatener mot_3 mot_3 in
string_of_mot mot_4 (fun x -> x);;
- : string = "abbabb"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
```

Exemple complet 1 : mots fonctionnels

On peut aussi définir des mots particuliers, comme les mots de Fibonacci.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun i -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun i -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

```
# let w = mot_fibo 0 in string_of_mot w (fun x -> x);;
- : string = "b"
# let w = mot_fibo 1 in string_of_mot w (fun x -> x);;
- : string = "a"
# let w = mot_fibo 2 in string_of_mot w (fun x -> x);;
- : string = "ab"
# let w = mot_fibo 3 in string_of_mot w (fun x -> x);;
- : string = "aba"
# let w = mot_fibo 4 in string_of_mot w (fun x -> x);;
- : string = "abaab"
# let w = mot_fibo 5 in string_of_mot w (fun x -> x);;
- : string = "abaababa"
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Exemple complet 2 : images fonctionnelles

Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y then
            {rouge = 127; vert = 127;
                bleu = 127}
        else
            {rouge = 255; vert = 255;
                bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

Exemple complet 2 : images fonctionnelles

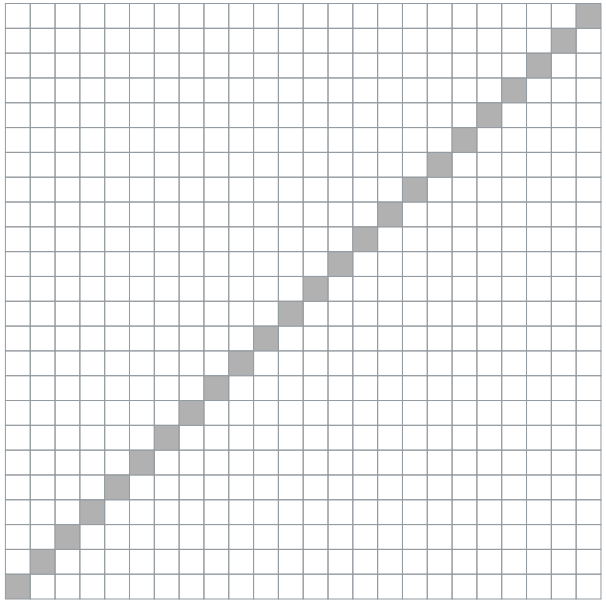
Reprenons la façon fonctionnelle de représenter les images vue précédemment :

```
type point = int * int
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
type couleur = {rouge : int; vert : int; bleu : int}
```

Par exemple,

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y then
            {rouge = 127; vert = 127;
                bleu = 127}
        else
            {rouge = 255; vert = 255;
                bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

représente l'image



Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Voici une fonction qui calcule l'image miroir d'une image en entrée :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x + 1, y)
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Exemple complet 2 : images fonctionnelles

Voici une fonction qui calcule l'image obtenue en complémentant les couleurs des pixels d'une image en entrée :

```
let complementer im =
    let contenus_pixels p =
        let (x, y) = p in
        {rouge = 255 - (im.contenus_pixels p).rouge;
         vert = 255 - (im.contenus_pixels p).vert;
         bleu = 255 - (im.contenus_pixels p).bleu}
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Voici une fonction qui calcule l'image miroir d'une image en entrée :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x + 1, y)
    in
    {contenus_pixels = contenus_pixels;
     largeur = im.largeur;
     hauteur = im.hauteur}
```

Toutes ces fonctions s'évaluent en temps $\Theta(1)$. Cette complexité ne dépend donc pas de la taille de l'image !

Exemple complet 3 : 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 3 : 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

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

où les α_i sont des entiers.

Exemple complet 3 : 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

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

où les α_i sont des entiers.

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

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

Exemple complet 3 : 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

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

où les α_i sont des entiers.

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

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

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

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

Exemple complet 3 : 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

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

où les α_i sont des entiers.

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

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

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

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

Question : comment représenter des séries génératrices ?

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_n de t^n.

C'est le type

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

Exemple complet 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_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 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_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 3 : séries génératrices

Réponse : par une fonction qui à tout entier positif n associe le coefficient α_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 3 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

Exemple complet 3 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :

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

Exemple complet 3 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\ ;$$
2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k\ ;$$

Exemple complet 3 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)+\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)t^k\ ;$$
2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\bullet\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k t^k\ ;$$
3. **produit** :
$$\left(\sum_{n\geqslant 0}\alpha_n t^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m t^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}t^k.$$

Exemple complet 3 : séries génératrices

Il existe plusieurs opérations sur les séries génératrices :

1. **somme** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) + \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} (\alpha_k + \beta_k) t^k \ ;$$
2. **produit d'Hadamard** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \bullet \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \alpha_k \beta_k t^k \ ;$$
3. **produit** :
$$\left(\sum_{n\geqslant 0} \alpha_n t^n\right) \cdot \left(\sum_{m\geqslant 0} \beta_m t^m\right) = \sum_{k\geqslant 0} \sum_{i=0}^{k} \alpha_i \beta_{k-i} t^k .$$

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

Exemple complet 3 : séries génératrices

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

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

Exemple complet 3 : 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 3 : 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 3 : 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 3 : 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 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire 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 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire 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 3 : séries génératrices

L'implantation du produit est un peu plus complexe dans sa mise en œuvre : elle utilise une fonction auxiliaire 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 3 : séries génératrices

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

Plan

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

Objets polymorphes

Un objet est dit polymorphe s'il n'est pas d'un type fixé.

- Une **fonction polymorphe** est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

- Un **type polymorphe** est un type paramétré. P.ex.,

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

- Une **valeur polymorphe** est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé. P.ex.,

```
# Vide;;
- : 'a liste = Vide
```