Fonctionnelles sur les listes

G. Dewaele

Lycée Louis-le-Grand

Objectif

On souhaite effectuer une opération sur tous les éléments d'une liste

En Python :

```python
lst = [ 1, 2, 3, 4, 5 ]

for elem in lst :
    foo(elem)
```

List.iter

En OCaml :

```
List.iter foo lst;;
```


Exécute « `foo elem` » sur tous les éléments `elem` de la liste `lst`

Les éléments sont traités de la gauche vers la droite

List.iter

Exemple : imprimer les éléments d'une liste d'entiers

```
# let lst = [ 1; 2; 3; 4; 5 ];;
val lst : int list = [1; 2; 3; 4; 5]

# List.iter print_int lst;;
12345- : unit = ()
```

List.iter

Pour une 'a list, la fonction doit être de type 'a -> unit

```
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Le résultat d'un appel à List.iter est toujours un unit

List.map

On peut vouloir la liste des résultats !

En Python, on utilise une compréhension :

```
[ foo(elem) for elem in lst ]
```

List.map

En OCaml :

```
List.map foo lst;;
```

Exécute « `foo elem` » sur tous les éléments `elem` de la liste `lst`

Construit une liste avec les résultats

$$[\ a_1;\ a_2;\ \ldots;\ a_n\ ] \mapsto [\ \texttt{foo}\ a_1;\ \texttt{foo}\ a_2;\ \ldots;\ \texttt{foo}\ a_n\ ]$$

L'ordre des évaluations n'est pas spécifié !

List.map

Si la fonction est de type `'a -> 'b` :

- on traite une `'a list`
- on obtient une `'b list` de même longueur

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

List.map

Quelques exemples...

```
# List.map int_of_float [ 1.0; 2.5; 3.14 ];;
- : int list = [1; 2; 3]
```

List.map

Quelques exemples...

```
# List.map int_of_float [ 1.0; 2.5; 3.14 ];;
- : int list = [1; 2; 3]
```



```
# let f n = float_of_int n ** 2.0;;
val f : int -> float = <fun>

# List.map f [ 1; 2; 3; 4; 5 ];;
- : float list = [1.; 4.; 9.; 16.; 25.]
```

List.map

Pour une fonction retournant des **unit** :

```
# List.map print_int [ 1; 2; 3; 4; 5 ];;
12345- : unit list = [(); (); (); (); ()]
```

Ici, les éléments ont été traités de gauche à droite

Ce n'est *pas* garanti !

List.map

On peut programmer son propre `List.map` :

```
# let rec map foo = function
    | [] -> []
    | t::q -> foo t :: (map foo q);;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```


L'ordre des évaluations n'est toujours pas spécifié !

On ne sait pas si OCaml effectue « `foo t` » ou « `map foo q` » d'abord

List.map

Pour garantir l'ordre de gauche à droite :


```
# let rec map foo = function
    | [] -> []
    | t::q -> let nouv_t = foo t in
                  nouv_t :: map foo q;;

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Retour sur `List.iter`

Pour implémenter soi-même `List.iter` :

```
# let rec iter foo = function
    | [] -> ()
    | t::q -> let _ = foo t in
                iter foo q;;

val iter : ('a -> 'b) -> 'a list -> unit = <fun>
```

Et au-dela ?

Pour sommer les éléments d'une liste d'entiers

En Python

```python
sum(lst)
```

ou bien

```python
def somme(lst) :
    acc = 0
    for elem in lst :
        acc = acc + elem
    return acc
```

Et au-dela ?

Pour sommer les éléments d'une liste d'entiers

En OCaml

```
let rec somme = function
  | [] -> 0
  | t::q -> t + somme q;;
```

Et au-dela ?

Pour déterminer le plus grand élément d'une liste d'entiers

En Python

```python
max(lst)
```

ou bien

```python
def maximum(lst) :
    plus_grand = lst[0]
    for elem in lst :
        if elem > plus_grand :
            plus_grand = elem
    return plus_grand
```

Et au-dela ?

Pour déterminer le plus grand élément d'une liste d'entiers

En OCaml

```
let rec maximum = function
  | [] -> failwith "Liste vide"
  | [ elem ] -> elem
  | t::q -> max t (maximum q);;
```

Et au-dela ?

Écrire une fonction récursive est généralement suffisant

C'est une construction très fréquente

On peut vouloir une formulation plus succinte !

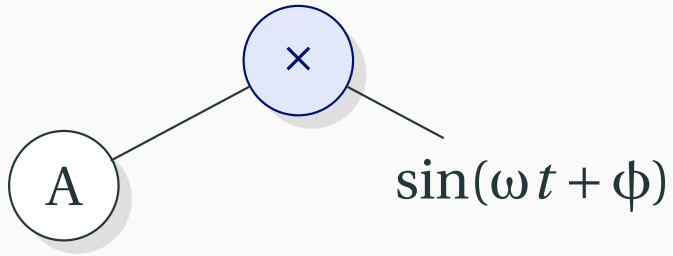
Arbres d'expressions

Considérons l'expression « $A\sin(\omega t + \phi)$ »

Arbres d'expressions

Considérons l'expression « $\mathrm{A}\sin(\omega t + \phi)$ »

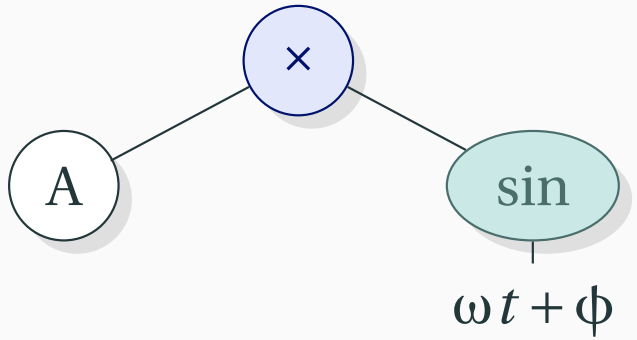
On peut l'interpréter sous forme arborescente :



Arbres d'expressions

Considérons l'expression « $\mathrm{A}\sin(\omega t + \phi)$ »

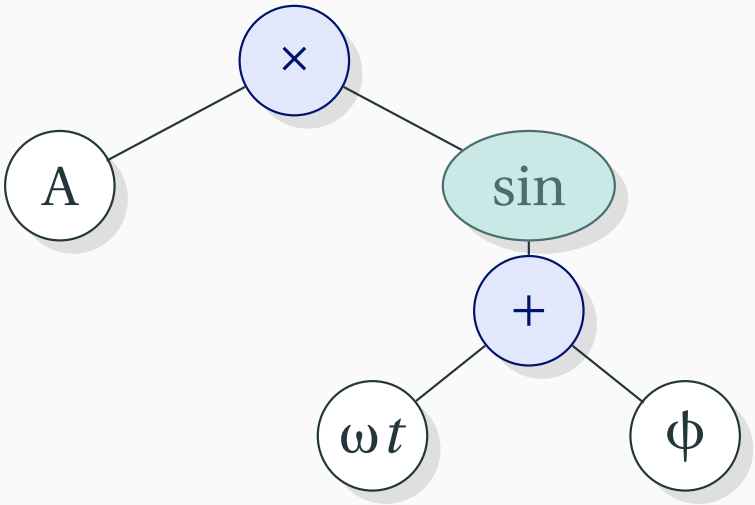
On peut l'interpréter sous forme arborescente :



Arbres d'expressions

Considérons l'expression « $\mathrm{A}\sin(\omega t + \phi)$ »

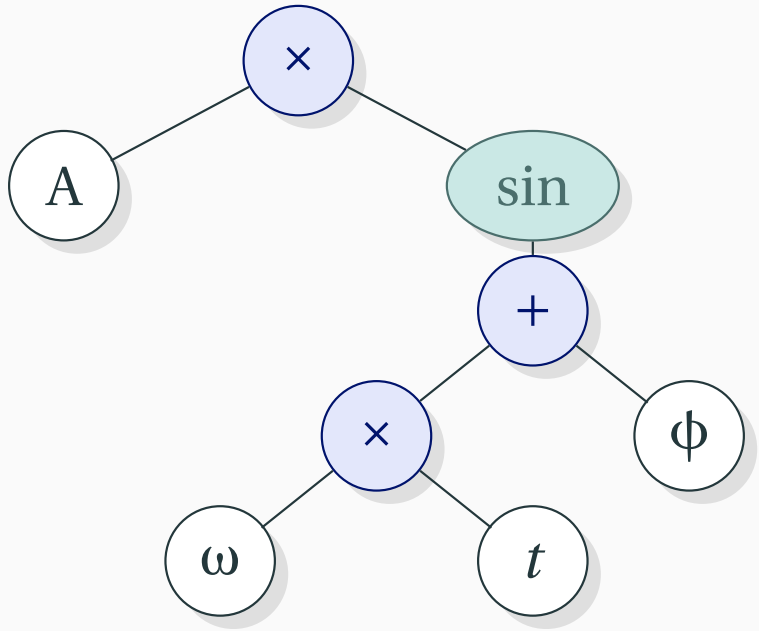
On peut l'interpréter sous forme arborescente :



Arbres d'expressions

Considérons l'expression « $\mathrm{A}\sin(\omega t + \phi)$ »

On peut l'interpréter sous forme arborescente :



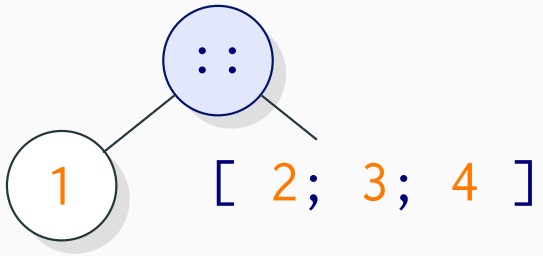
Cas d'une liste

Pour une liste `[ 1; 2; 3; 4 ]`

Cas d'une liste

Pour une liste `[ 1; 2; 3; 4 ]`

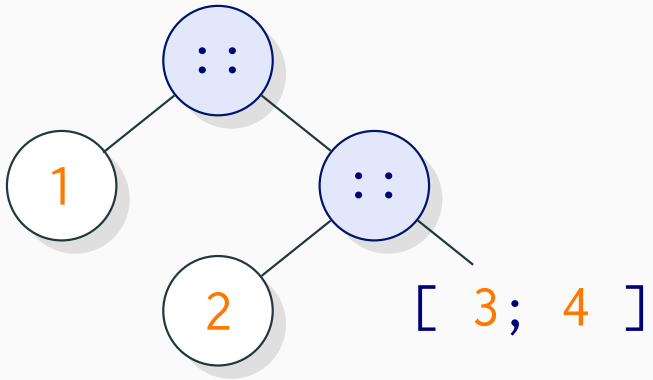
On peut également l'interpréter sous forme arborescente :



Cas d'une liste

Pour une liste `[ 1; 2; 3; 4 ]`

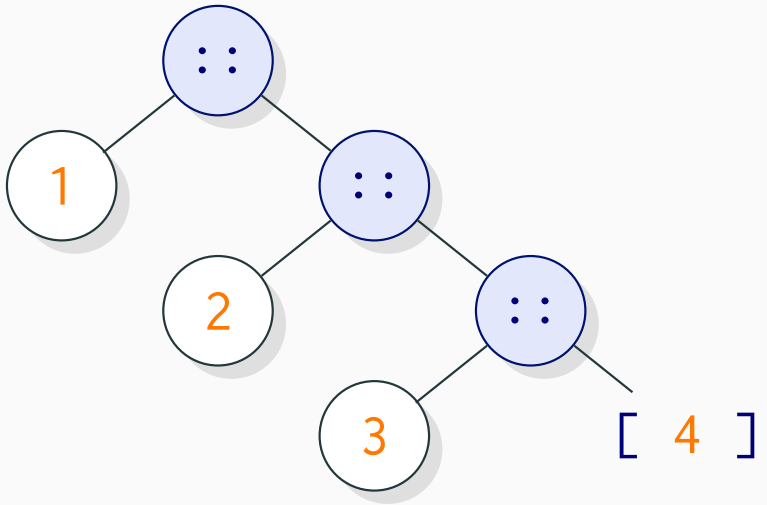
On peut également l'interpréter sous forme arborescente :



Cas d'une liste

Pour une liste `[ 1; 2; 3; 4 ]`

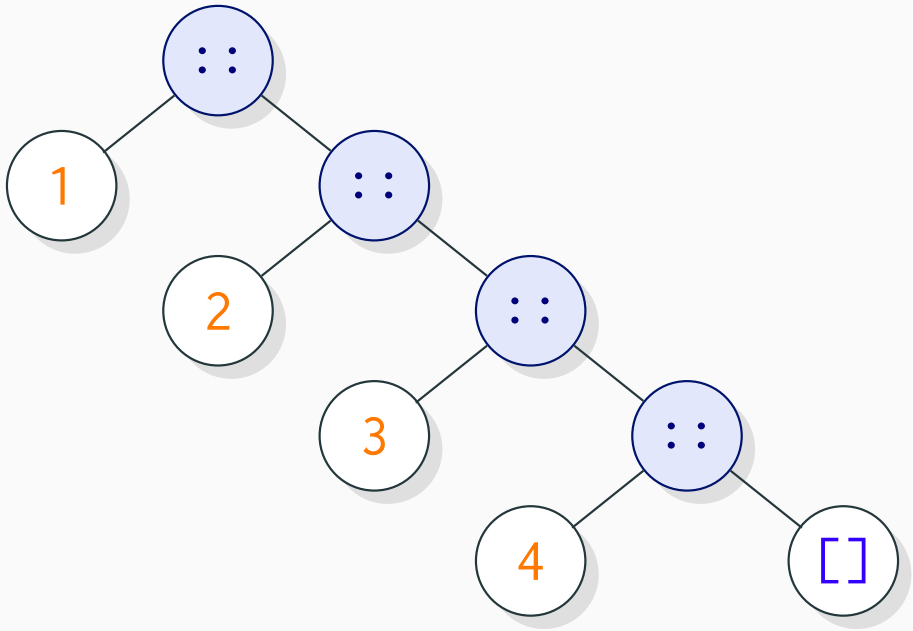
On peut également l'interpréter sous forme arborescente :



Cas d'une liste

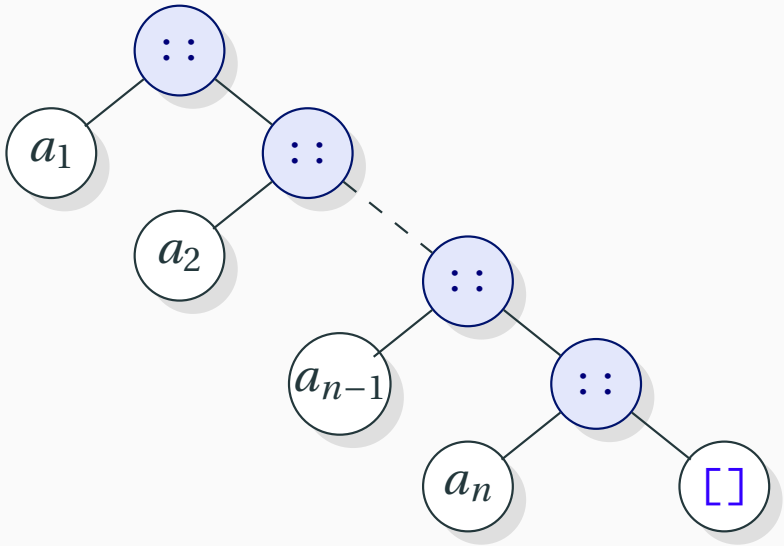
Pour une liste `[ 1; 2; 3; 4 ]`

On peut également l'interpréter sous forme arborescente :



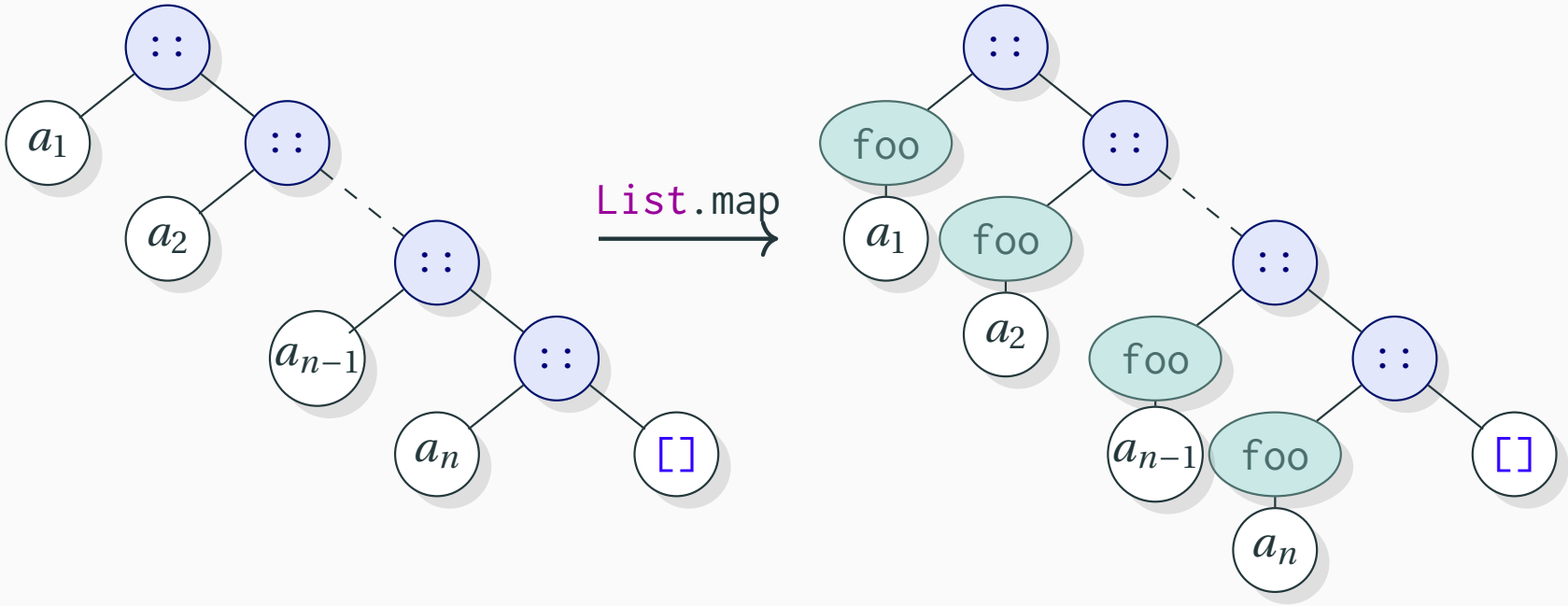
Que fait `List.map` ?

`List.map foo` « insère » pour chaque élément un appel à `foo` :



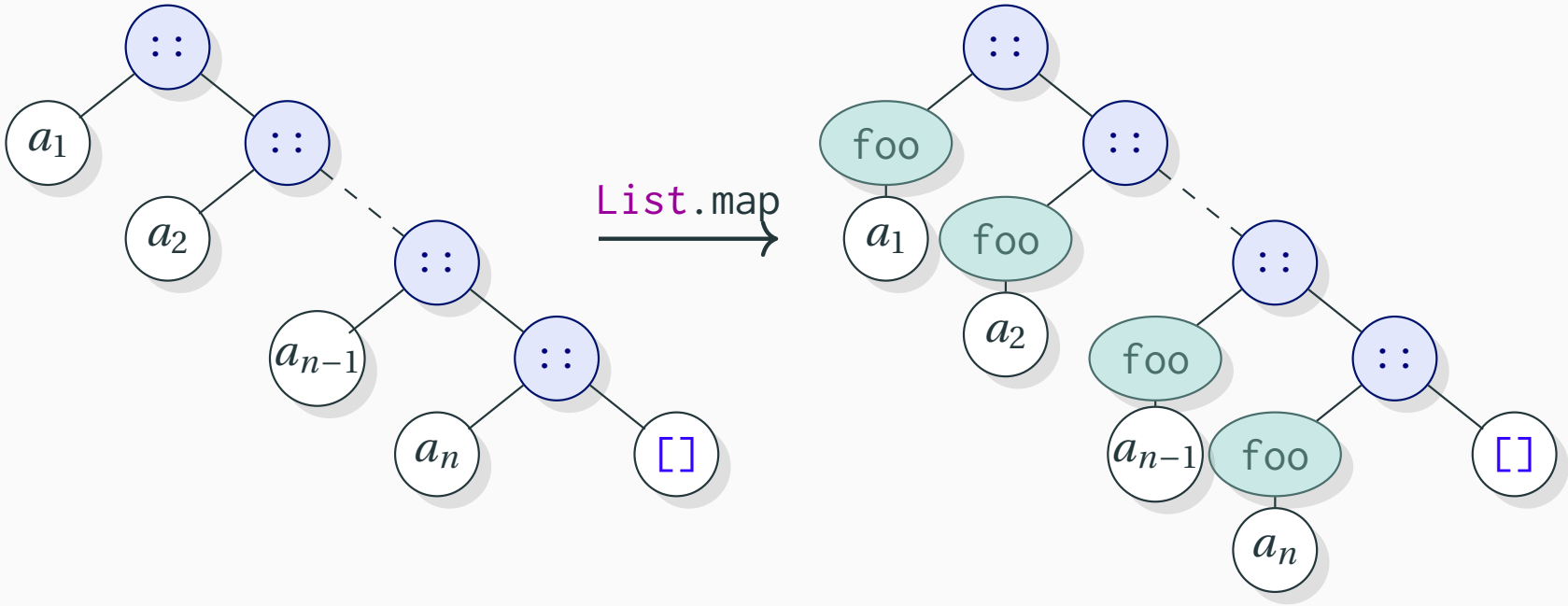
Que fait `List.map` ?

`List.map foo` « insère » pour chaque élément un appel à `foo` :



Que fait `List.map` ?

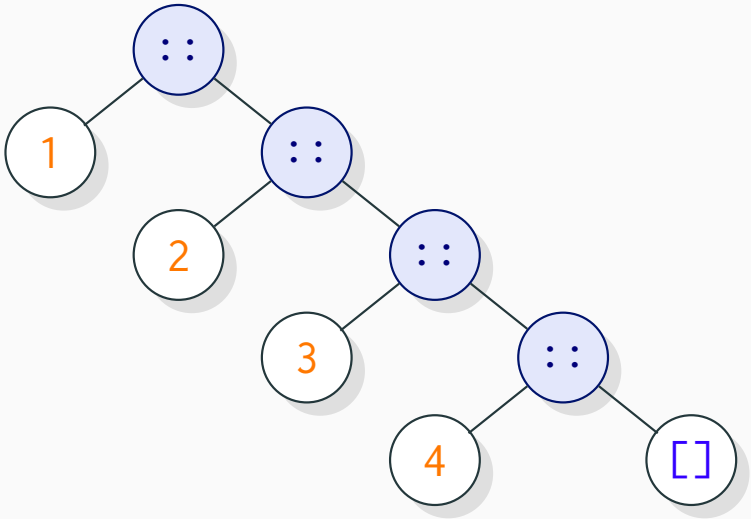
`List.map foo` « insère » pour chaque élément un appel à `foo` :



Si `foo` est de signature `'a -> 'b`, une `'a list` devient `'b list` !

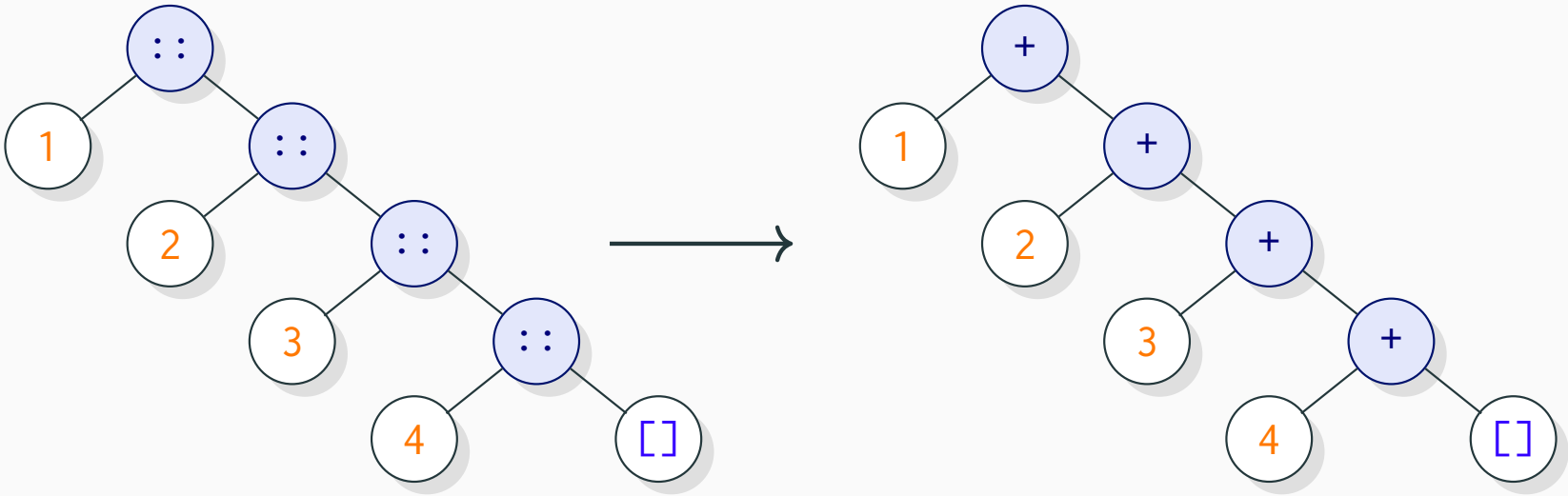
Somme des termes d'une liste

Pour effectuer la somme des éléments d'une liste :



Somme des termes d'une liste

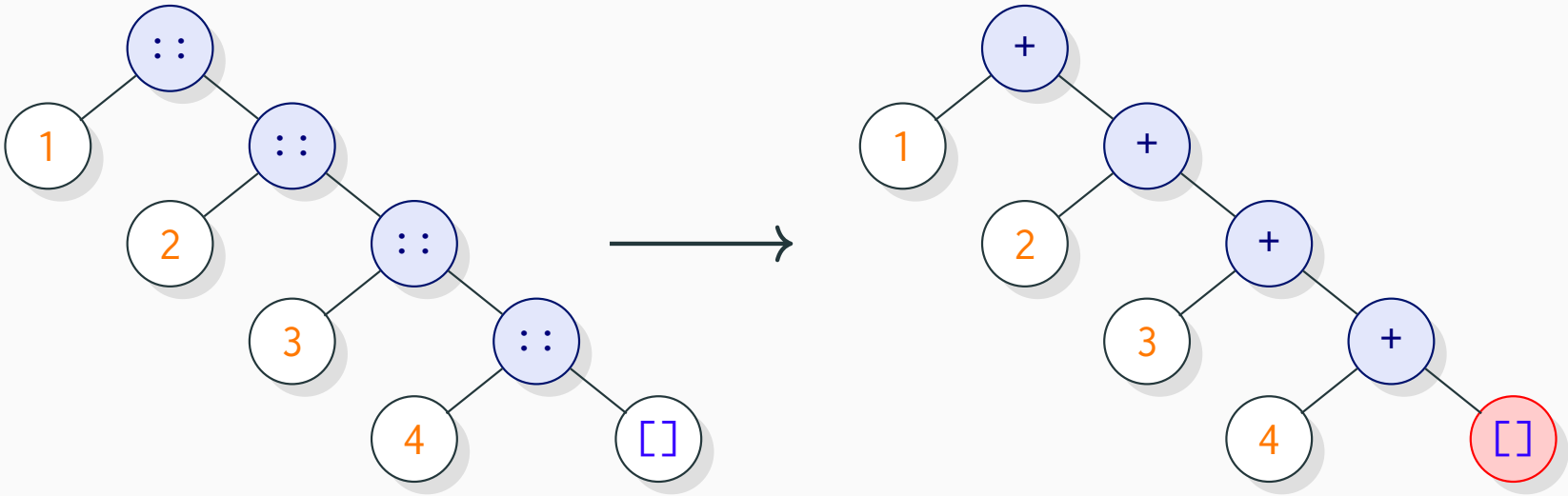
Pour effectuer la somme des éléments d'une liste :



On veut remplacer les : : par une fonction à deux arguments...

Somme des termes d'une liste

Pour effectuer la somme des éléments d'une liste :

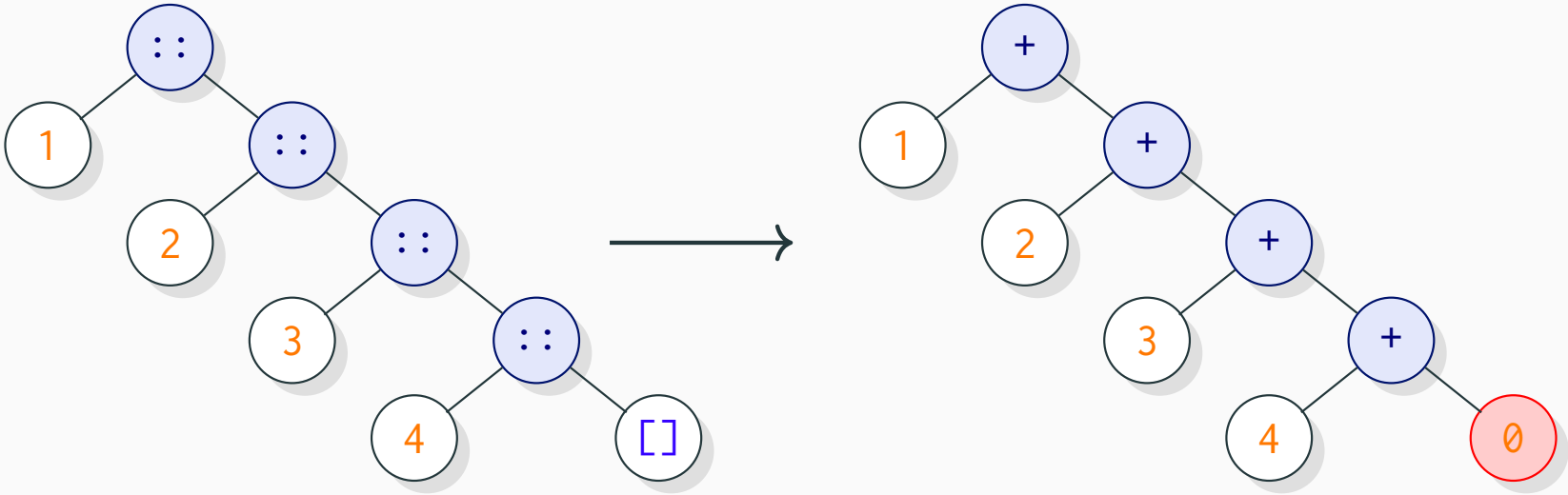


On veut remplacer les `::` par une fonction à deux arguments...

et `[]` par autre chose !

Somme des termes d'une liste

Pour effectuer la somme des éléments d'une liste :

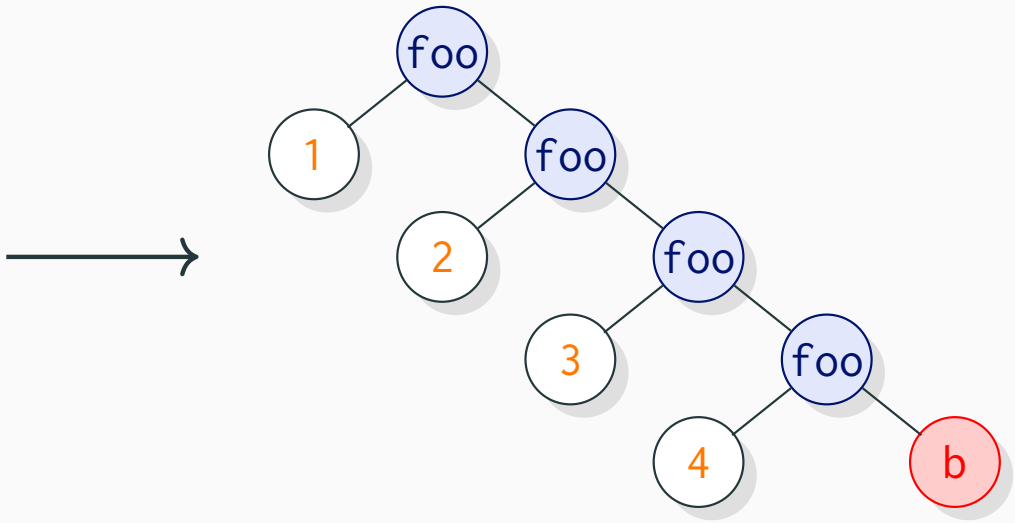


On veut remplacer les `::` par une fonction à deux arguments...

et `[]` par autre chose !

Somme des termes d'une liste

Plus généralement :

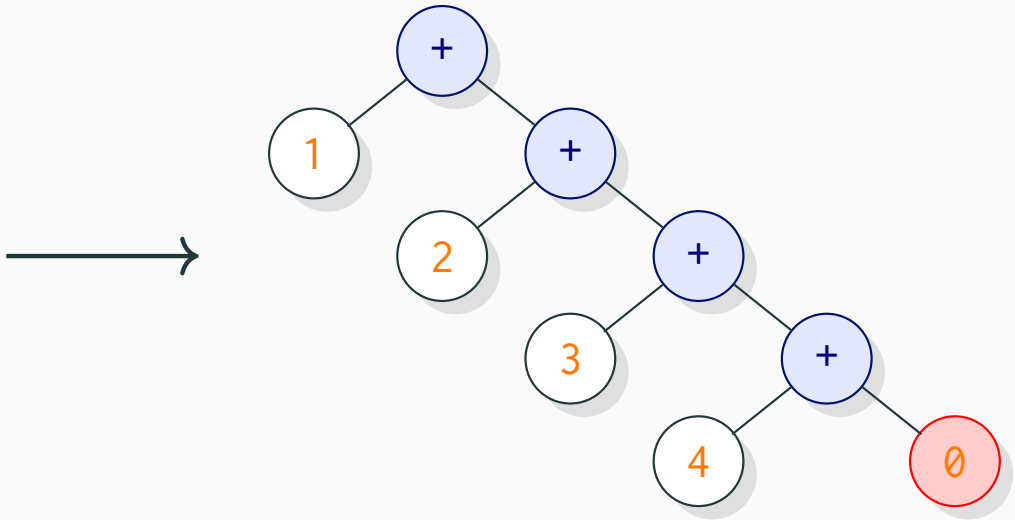


Pour une `'a list`, si on veut un résultat de type `'b` :

- la fonction remplaçant `::` doit être de type `'a -> 'b -> 'b`
- ce qui remplace `[]` de type `'b`

Somme des termes d'une liste

Pour la somme des éléments d'une liste :



Pour la somme des éléments, `'a` et `'b` sont des `int`

- la fonction est `fun a b -> a+b`
- ce qui remplace `[]` est `0`

Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

C'est une fonction qui prend trois arguments :

- `('a -> 'b -> 'b)` est la fonction qui remplace `::`
- `'a list` est la liste que l'on traite
- le premier `'b` désigne l'élément qui remplace `[]`

Enfin, le second `'b` indique le type du résultat

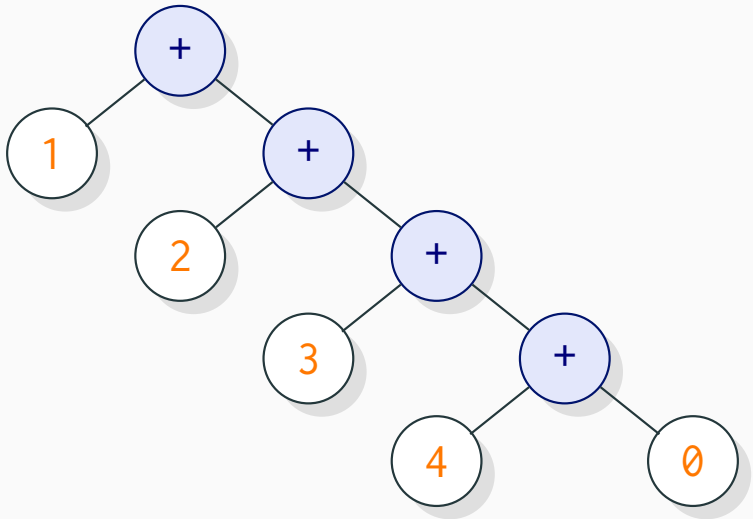
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Le nom de la fonction décrit bien l'opération :

- « `fold_` » car il s'agit de « replier » l'arbre
- « `_right` » car les repliements sont effectués à droite



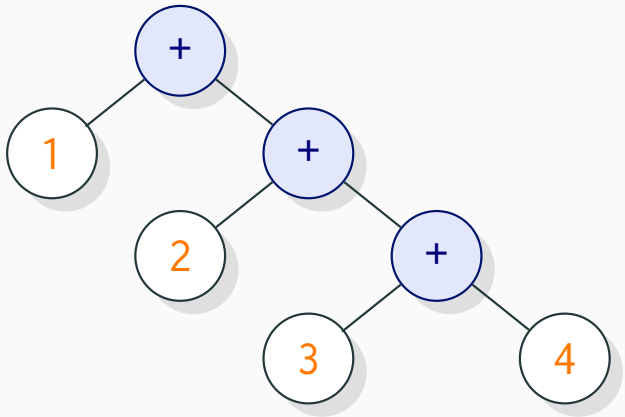
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Le nom de la fonction décrit bien l'opération :

- « `fold_` » car il s'agit de « replier » l'arbre
- « `_right` » car les repliements sont effectués à droite



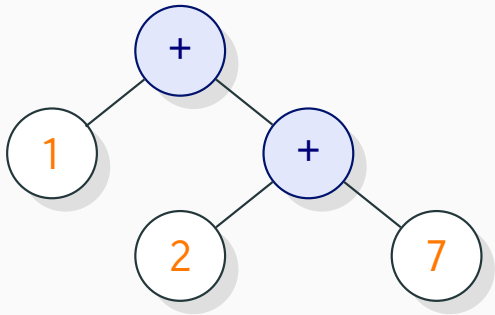
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Le nom de la fonction décrit bien l'opération :

- « `fold_` » car il s'agit de « replier » l'arbre
- « `_right` » car les repliements sont effectués à droite



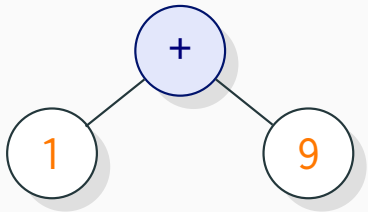
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Le nom de la fonction décrit bien l'opération :

- « `fold_` » car il s'agit de « replier » l'arbre
- « `_right` » car les repliements sont effectués à droite



Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Le nom de la fonction décrit bien l'opération :

- « `fold_` » car il s'agit de « replier » l'arbre
- « `_right` » car les repliements sont effectués à droite

10

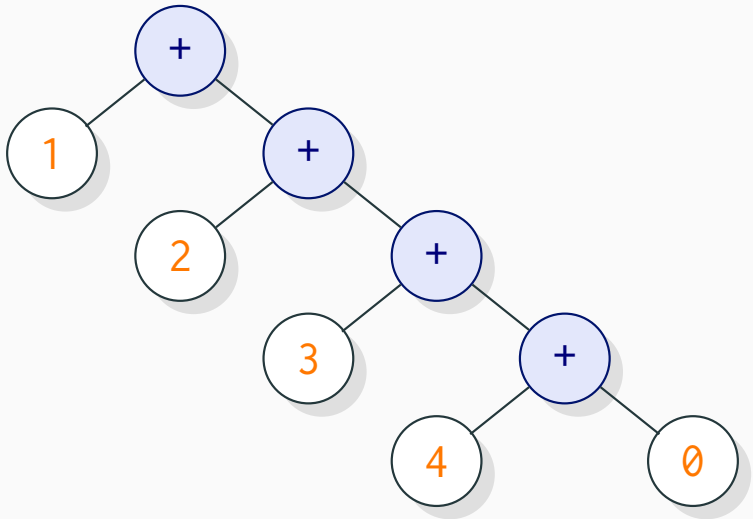
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, l'élément remplaçant `[]` à droite



Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite

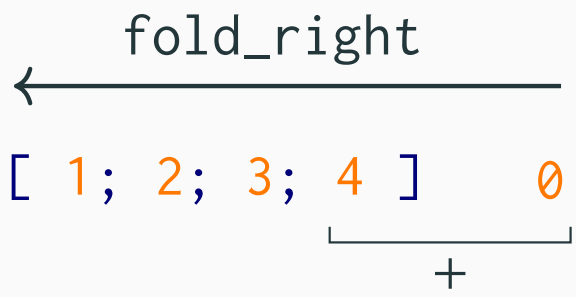
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite



Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite

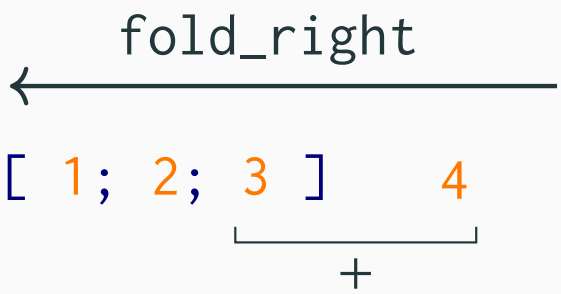
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite



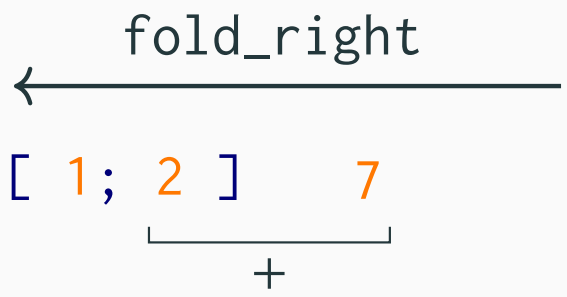
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite



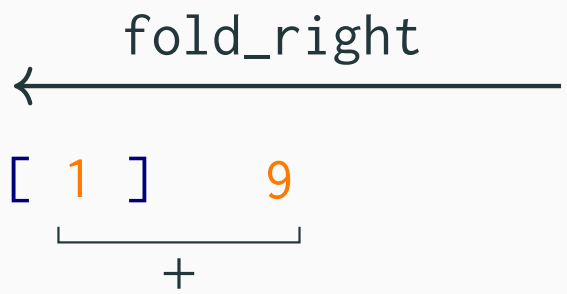
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite



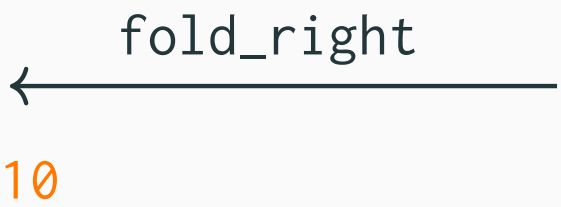
Somme des termes d'une liste

On utilise `List.fold_right` pour effectuer cette opération

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Les arguments respectent l'image arborescente :

- les arguments de la fonction utilisée sont dans l'ordre
- la liste est à gauche, le point de départ du repliement à droite



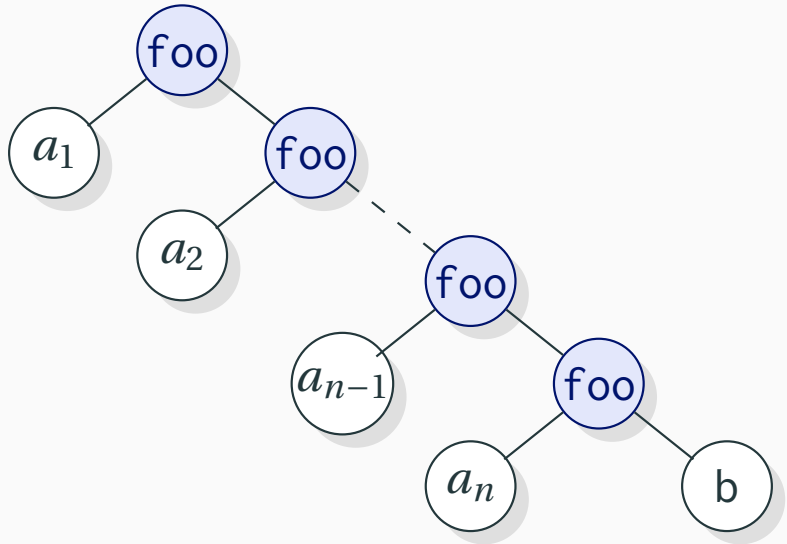
Somme des termes d'une liste

L'instruction

$$\texttt{List.fold\_right foo [} a_1\texttt{; } a_2\texttt{; ...; } a_{n-1}\texttt{; } a_n \texttt{] b}$$

Correspond donc à

$$\texttt{foo } a_1 \texttt{ (foo } a_2 \texttt{ (... (foo } a_{n-1} \texttt{ (foo } a_n \texttt{ b)) ...))}$$



Somme des termes d'une liste

On peut donc écrire :


```
# let somme a b = a + b;;
val somme : int -> int -> int = <fun>

# List.fold_right somme [ 1; 2; 3; 4 ] 0;;
- : int = 10
```

Somme des termes d'une liste

On peut donc écrire :

```
# let somme a b = a + b;;
val somme : int -> int -> int = <fun>

# List.fold_right somme [ 1; 2; 3; 4 ] 0;;
- : int = 10
```

Ou plus directement :

```
# List.fold_right (fun a b -> a+b) [ 1; 2; 3; 4 ] 0;;
- : int = 10
```

Somme des termes d'une liste

Si l'on veut construire une fonction effectuant cette somme :

```
# let somme_liste lst =
    List.fold_right (fun a b -> a+b) lst 0;;

val somme_liste : int list -> int = <fun>
```

Somme des termes d'une liste

Si l'on veut construire une fonction effectuant cette somme :

```
# let somme_liste lst =
    List.fold_right (fun a b -> a+b) lst 0;;

val somme_liste : int list -> int = <fun>
```

La fonction ainsi créée peut sommer une liste d'entiers :

```
# somme_liste [ 1; 2; 3; 4 ];;
- : int = 10
```

Somme des termes d'une liste

Une dernière chose...

Pour un opérateur binaire tel que `+`

`(+)` est un raccourci pour la fonction `(fun a b -> a+b)`

Somme des termes d'une liste

Une dernière chose...

Pour un opérateur binaire tel que `+`

`(+)` est un raccourci pour la fonction `(fun a b -> a+b)`

De sorte qu'on peut simplement écrire

```
# List.fold_right (+) [ 1; 2; 3; 4 ] 0;;
- : int = 10
```

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

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

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1
- On replie avec la fonction « produit »

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1
- On replie avec la fonction « produit »

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1
- On replie avec la fonction « produit »

Produit des éléments d'une liste d'entiers

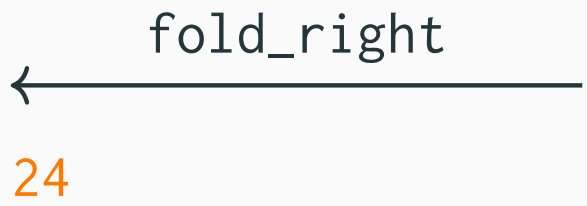
Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1
- On replie avec la fonction « produit »

Produit des éléments d'une liste d'entiers

Pour calculer le produit de tous les termes d'une liste :

- On part de la valeur 1
- On replie avec la fonction « produit »



Produit des éléments d'une liste d'entiers

Cela peut s'écrire :

```
# let produit_liste lst =
    List.fold_right (fun a b -> a * b) lst 1;;

val produit_liste : int list -> int = <fun>
```

Ou bien

```
# let produit_liste lst =
    List.fold_right ( * ) lst 1;;
```

```
# produit_liste [ 1; 2; 3; 4 ];;
- : int = 24
```

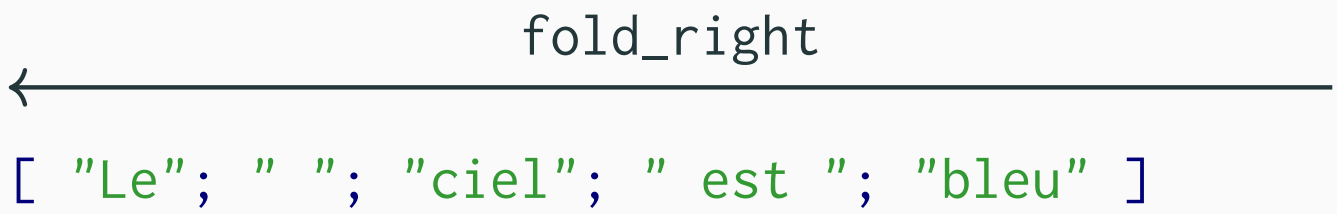
Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

```
[ "Le"; " "; "ciel"; " est "; "bleu" ]
```

Concaténer une liste de chaînes de caractères

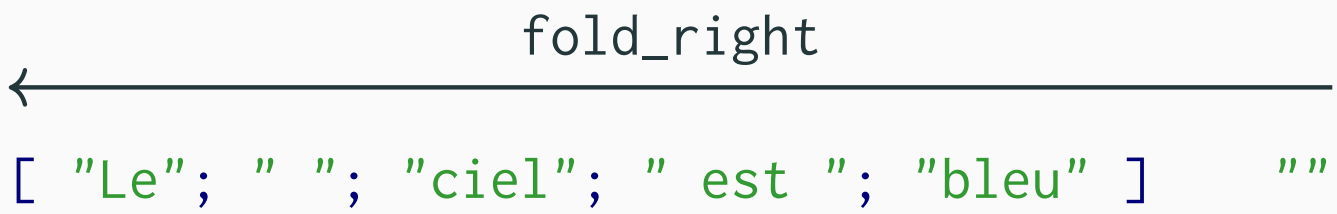
Pour concaténer des chaînes de caractères dans une liste :



Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

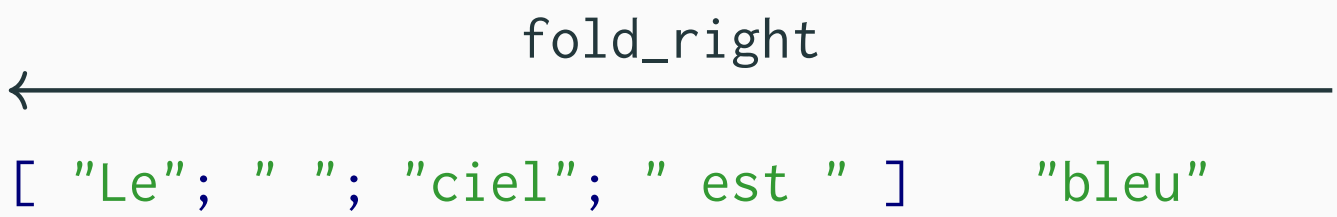
- On part d'une chaîne vide `""`
- On replie avec des concaténations



Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

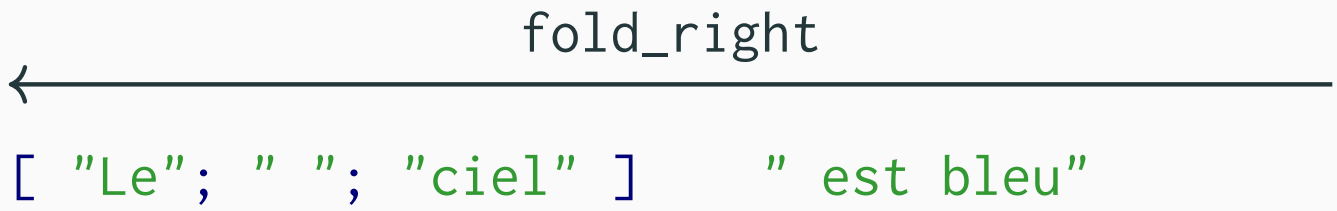
- On part d'une chaîne vide `""`
- On replie avec des concaténations



Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

- On part d'une chaîne vide `""`
- On replie avec des concaténations



Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

- On part d'une chaîne vide `""`
- On replie avec des concaténations

Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

- On part d'une chaîne vide `""`
- On replie avec des concaténations

Concaténer une liste de chaînes de caractères

Pour concaténer des chaînes de caractères dans une liste :

- On part d'une chaîne vide `""`
- On replie avec des concaténations

Concaténer une liste de chaînes de caractères

Cela peut s'écrire :

```
# let concat lst =
    List.fold_right (fun a b -> a ^ b) lst "";;

val concat : string list -> string = <fun>
```

Ou bien

```
# let concat lst =
    List.fold_right (^) lst "";;
```

```
# concat [ "Le"; " "; "ciel"; " est "; "bleu" ];;
- : string = "Le ciel est bleu"
```

Concaténer une liste de chaînes de caractères

Dans ce dernier cas, on effectue $n-1$ concaténations

C'est inutilement coûteux !

Concaténer une liste de chaînes de caractères

Dans ce dernier cas, on effectue $n-1$ concaténations

C'est inutilement coûteux !

Il existe une fonction pour cela :

```
# String.concat;;
- : string -> string list -> string = <fun>
```

Le premier argument est glissé entre chaque chaîne de la liste :

```
# String.concat " " [ "Le"; "ciel"; "est"; "bleu" ];;
- : string = "Le ciel est bleu"
```

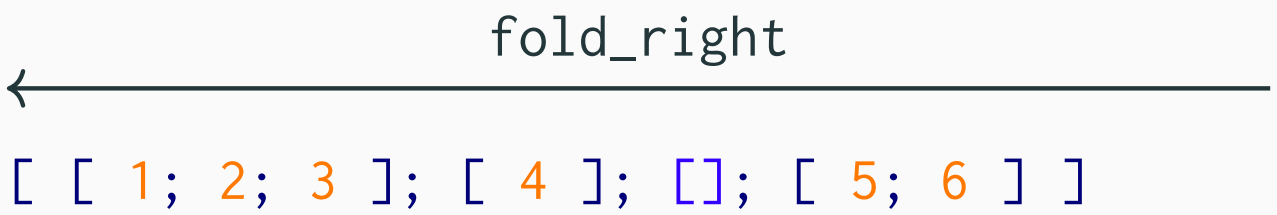
Aplatir une liste de listes

Pour aplatir une liste de liste :

```
[ [ 1; 2; 3 ]; [ 4 ]; []; [ 5; 6 ] ]
```

Aplatir une liste de listes

Pour aplatir une liste de liste :



Aplatir une liste de listes

Pour aplatir une liste de liste :

- On part d'une liste vide `[]`
- On replie avec des concaténations

Aplatir une liste de listes

Pour aplatir une liste de liste :

- On part d'une liste vide `[]`
- On replie avec des concaténations

Aplatir une liste de listes

Pour aplatir une liste de liste :

- On part d'une liste vide `[]`
- On replie avec des concaténations

Aplatir une liste de listes

Pour aplatir une liste de liste :

- On part d'une liste vide `[]`
- On replie avec des concaténations

Aplatir une liste de listes

Pour aplatir une liste de liste :

- On part d'une liste vide `[]`
- On replie avec des concaténations

Aplatir une liste de listes

Cela peut s'écrire :

```
# let aplatir lst =
    List.fold_right (fun a b -> a @ b) lst [];;

val aplatir : 'a list list -> 'a list = <fun>
```

Ou bien

```
# let aplatir lst =
    List.fold_right (@) lst [];;
```

```
# aplatir [ [ 1; 2; 3 ]; [ 4 ]; []; [ 5; 6 ] ];;
- : int list = [1; 2; 3; 4; 5; 6]
```

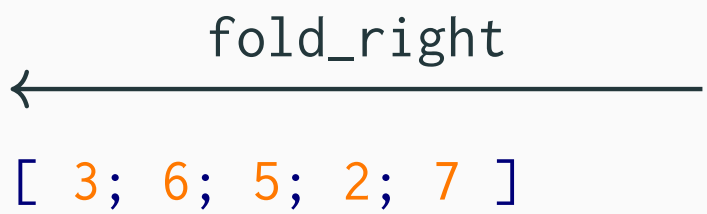
Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

```
[ 3; 6; 5; 2; 7 ]
```

Supprimer les valeurs paires

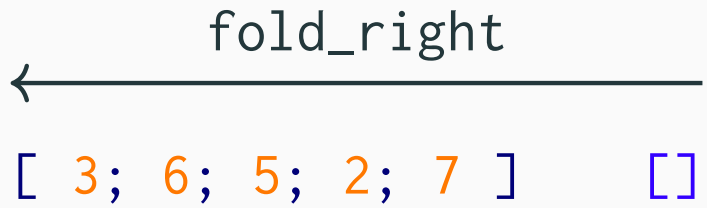
Pour supprimer les éléments pairs dans une liste :



Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*



Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*

Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*

Supprimer les valeurs paires

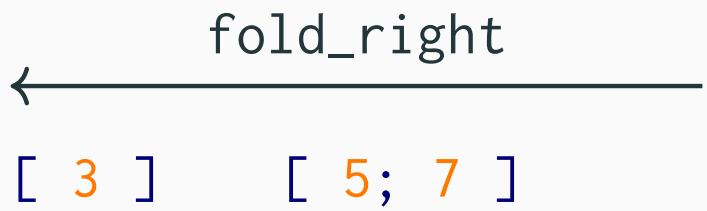
Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*

Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*



Supprimer les valeurs paires

Pour supprimer les éléments pairs dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont impairs*

Supprimer les valeurs paires

Cette fois-ci, on a besoin d'une fonction spécifique !


```
# let conse_si_impair elem lst =
    match elem mod 2 with
      | 0 -> lst
      | _ -> elem::lst;;

val conse_si_impair : int -> int list
                        -> int list = <fun>
```

Supprimer les valeurs paires

Ensuite, on peut écrire notre fonction :

```
# let supprime_pairs lst =
    List.fold_right conse_si_impair lst [];;

val supprime_pairs : int list -> int list = <fun>
```

```
# supprime_pairs [ 3; 6; 5; 2; 7 ];;
- : int list = [3; 5; 7]
```

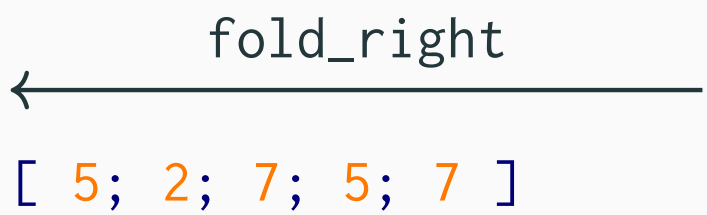
Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

```
[ 5; 2; 7; 5; 7 ]
```

Supprimer les doublons

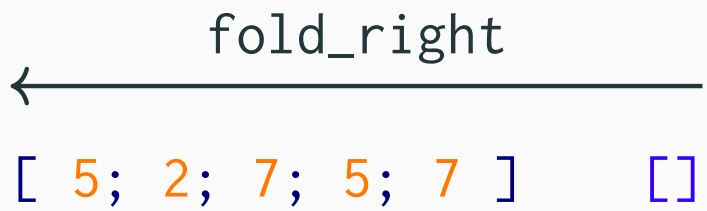
Pour supprimer les éléments en double dans une liste :



Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

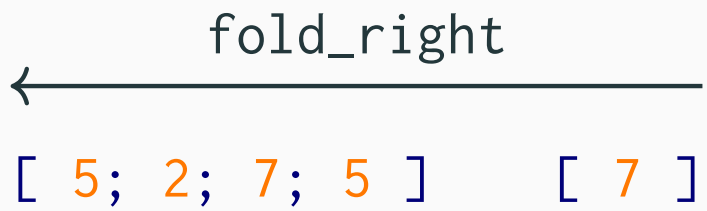
- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*



Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*



Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*

Supprimer les doublons

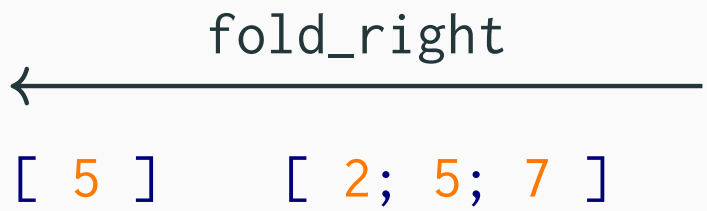
Pour supprimer les éléments en double dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*

Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*



Supprimer les doublons

Pour supprimer les éléments en double dans une liste :

- On part d'une liste vide `[]`
- On ajoute (avec `::`) les éléments *s'ils sont absents*

Supprimer les doublons

Cette fois encore, on a besoin d'une fonction spécifique :

```
# let conse_si_absent elem = function
    | lst when List.mem elem lst -> lst
    | lst -> elem::lst;;

val conse_si_absent : 'a -> 'a list -> 'a list = <fun>
```

Supprimer les doublons

Ensuite, on peut écrire notre fonction :

```
# let supprime_doublons lst =
    List.fold_right conse_si_absent lst [];;

val supprime_doublons : 'a list -> 'a list = <fun>
```



```
# supprime_doublons [ 5; 2; 7; 5; 7 ];;
- : int list = [2; 5; 7]
```

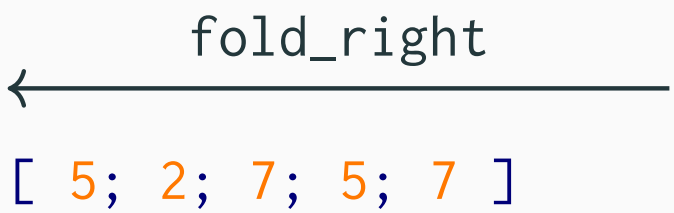
Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

```
[ 5; 2; 7; 5; 7 ]
```

Calculer la longueur d'une liste

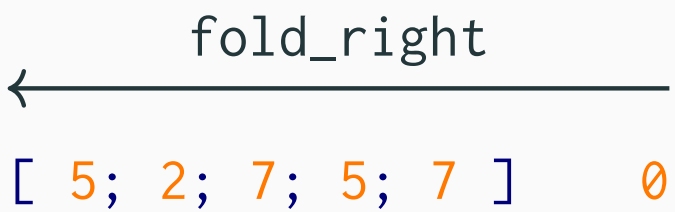
Pour déterminer le nombre d'éléments dans une liste :



Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

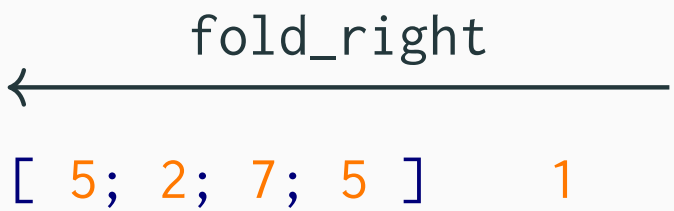
- On part de `0`
- On incrémente, *quel que soit l'élément extrait*



Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

- On part de `0`
- On incrémente, *quel que soit l'élément extrait*



Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

- On part de `0`
- On incrémente, *quel que soit l'élément extrait*

Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

- On part de `0`
- On incrémente, *quel que soit l'élément extrait*

Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

- On part de `0`
- On incrémente, *quel que soit l'élément extrait*

Calculer la longueur d'une liste

Pour déterminer le nombre d'éléments dans une liste :

- On part de `0`
- On incrémente, *quel que soit l'élément extrait*

Calculer la longueur d'une liste

Si l'on veut construire une fonction effectuant cette somme :

```
# let longueur lst =
    List.fold_right (fun a b -> b+1) lst 0;;

val longueur : 'a list -> int = <fun>
```

Calculer la longueur d'une liste

Si l'on veut construire une fonction effectuant cette somme :

```
# let longueur lst =
    List.fold_right (fun a b -> b+1) lst 0;;

val longueur : 'a list -> int = <fun>
```

La fonction retourne bien la longueur de la liste :

```
# longueur [ 5; 2; 7; 5; 7 ];;
- : int = 5
```

Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

```
[ 5; 2; 7; 6; 4 ]
```

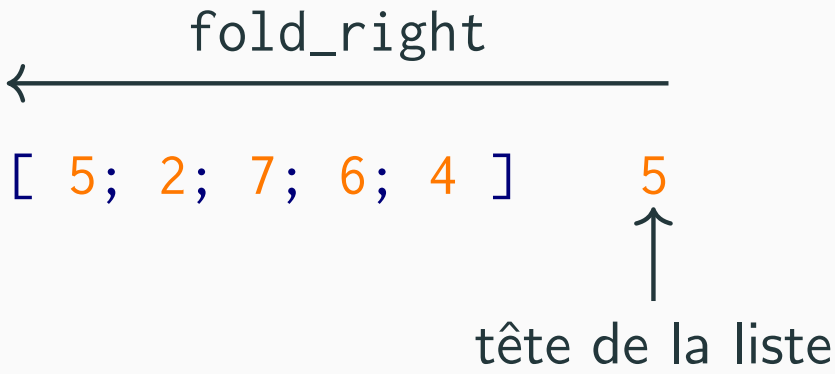
Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

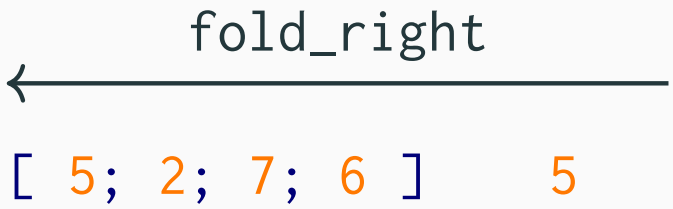
- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`



Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`



Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`

Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`

Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`

Déterminer le plus grand élément d'une liste

Pour déterminer le plus grand élément d'une liste :

- On part d'*un élément présent dans la liste*
- On replie avec la fonction `max`

Déterminer le plus grand élément d'une liste

Pour construire une fonction trouvant le plus grand élément :


```
# let maximum lst =
    List.fold_right max lst (List.hd lst);;

val maximum : 'a list -> 'a = <fun>
```

Déterminer le plus grand élément d'une liste

Pour construire une fonction trouvant le plus grand élément :

```
# let maximum lst =
    List.fold_right max lst (List.hd lst);;

val maximum : 'a list -> 'a = <fun>
```

Ou, afin d'éviter une comparaison inutile avec la tête :

```
# let maximum = function
    | [] -> failwith "Liste vide"
    | t::q -> List.fold_right max q t;;

val maximum : 'a list -> 'a = <fun>
```

Comment fonctionne `List.fold_right` ?

On cherche à calculer l'expression :

$$\texttt{foo } a_1 \texttt{ (foo } a_2 \texttt{ (... (foo } a_{n-1} \texttt{ (foo } a_n \texttt{ b)) ...))}$$

Cela peut s'écrire :


```
# let rec fold_right foo lst b =
    match lst with
      | [] -> b
      | t::q -> foo t (fold_right foo q b);;

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b
                   -> 'b = <fun>
```

Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

```
0    [ 1; 2; 3; 4 ]
```

Et dans le sens contraire ? `List.fold_left`

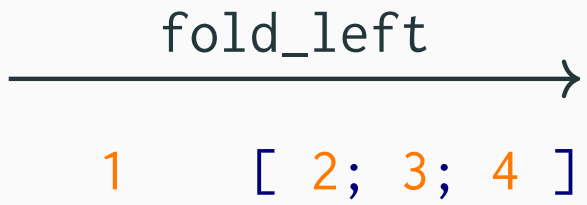
On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :



Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

Et dans le sens contraire ? `List.fold_left`

On peut vouloir effectuer la même chose dans le sens contraire !

Par exemple, pour la somme des éléments d'une liste :

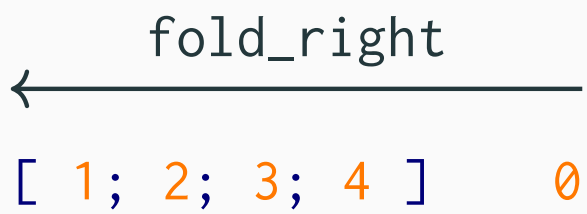
La fonction existe bien :

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Et dans le sens contraire ? `List.fold_left`

On remarquera la différence dans les signatures :

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```




```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

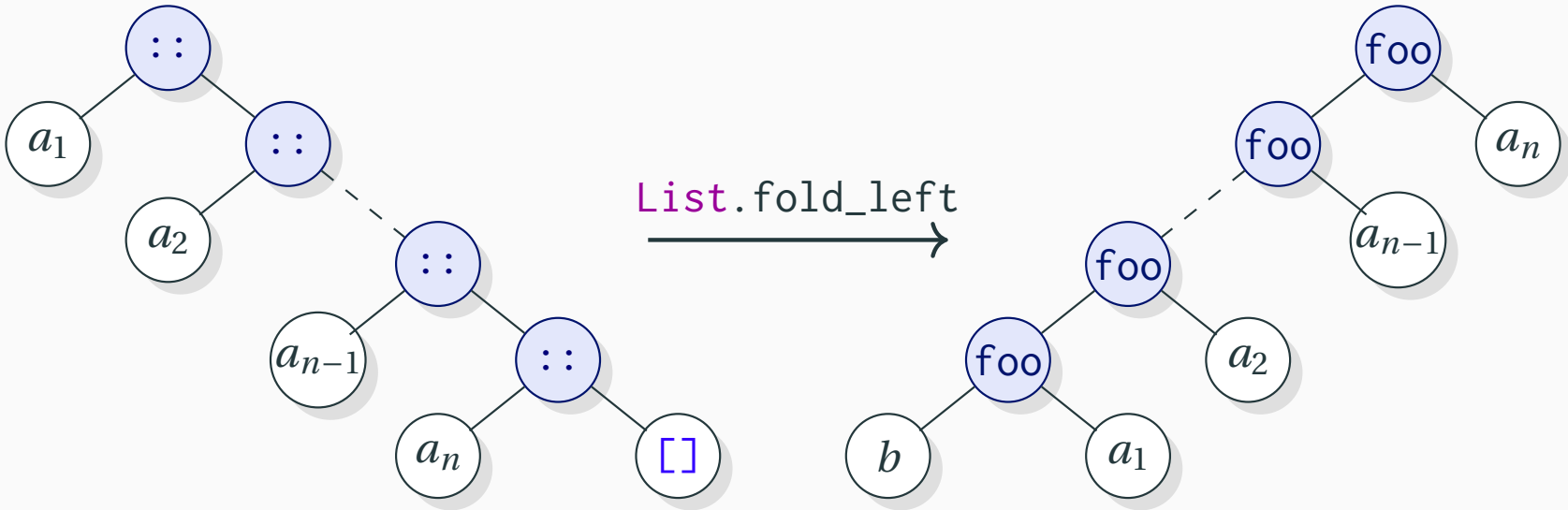
Et dans le sens contraire ? `List.fold_left`

L'instruction

$$\texttt{List.fold\_left foo b [} a_1\texttt{; } a_2\texttt{; ...; } a_{n-1}\texttt{; } a_n \texttt{]}$$

Correspond donc à

$$\texttt{foo (foo (... (foo (foo b } a_1\texttt{) } a_2\texttt{) ...) } a_{n-1}\texttt{) } a_n$$



Et dans le sens contraire ? `List.fold_left`

On peut réécrire certaines des fonctions précédentes :

```
# let somme_liste lst =
    List.fold_left (fun b a -> b+a) 0 lst;;

val somme_liste : int list -> int = <fun>
```

```
# let longueur lst =
    List.fold_left (fun b a -> b+1) 0 lst;;

val longueur : 'a list -> int = <fun>
```

Attention à l'ordre des arguments !

Et dans le sens contraire ? `List.fold_left`

L'argument de `somme_liste` est le dernier argument de `fold_left` :

```
# let somme_liste lst =
    List.fold_left (+) 0 lst;;

val somme_liste : int list -> int = <fun>
```

On peut donc simplifier les choses avec une application partielle :

```
# let somme_liste =
    List.fold_left (+) 0;;

val somme_liste : int list -> int = <fun>
```

Et dans le sens contraire ? `List.fold_left`

L'application partielle peut parfois jouer de mauvais tours :

```
# let longueur =
    List.fold_left (fun b a -> b+1) 0;;

val longueur : '_weak1 list -> int = <fun>
```


La fonction précédente n'est pas réellement polymorphe

La première utilisation fixera le type des éléments de la liste !

Et dans le sens contraire ? List.fold_left


```
# longueur;;
- : '_weak1 list -> int = <fun>

# longueur [ 1; 2; 3; 4 ];;
- : int = 4

# longueur;;
- : int list -> int = <fun>

# longueur [ 4.9; 10.23; 22.11 ];;
Characters 21-24: longueur [ 4.9; 10.23; 22.11 ];;
                                ^^^
Error: This expression has type float
       but an expression was expected of type int
```

Et dans le sens contraire ? `List.fold_left`

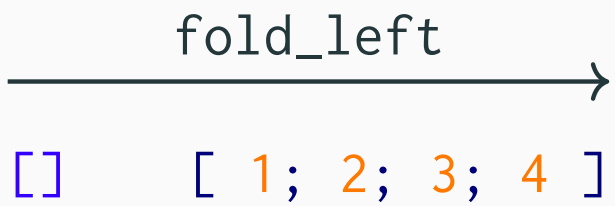
Avoir deux fonctions est intéressant si `foo` n'est pas commutative

Si les deux sont possibles, `List.fold_left` est à préférer

(Aucune importance pour les concours)

Et dans le sens contraire ? `List.fold_left`

Un exemple intéressant : `fun b a -> a::b`

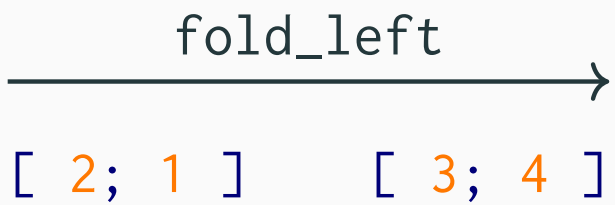


Et dans le sens contraire ? `List.fold_left`

Un exemple intéressant : `fun b a -> a::b`

Et dans le sens contraire ? `List.fold_left`

Un exemple intéressant : `fun b a -> a::b`

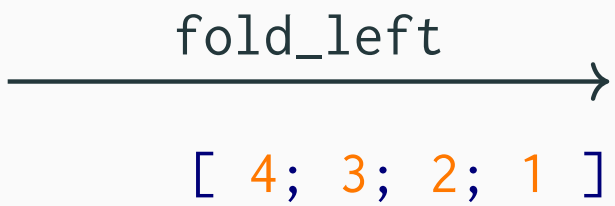


Et dans le sens contraire ? `List.fold_left`

Un exemple intéressant : `fun b a -> a::b`

Et dans le sens contraire ? `List.fold_left`

Un exemple intéressant : `fun b a -> a::b`



Retourner une liste

On peut donc écrire simplement une fonction retournant une liste :

```
# let retourne lst =
    List.fold_left (fun b a -> a::b) [] lst;;

val retourne : 'a list -> 'a list = <fun>
```


Il existe une fonction `List.rev`

Comment fonctionne `List.fold_left` ?

On cherche à calculer l'expression :

$$\texttt{foo (foo (... (foo (foo b } a_1\texttt{) } a_2\texttt{) ...) } a_{n-1}\texttt{) } a_n$$

Cela peut s'écrire :


```
# let rec fold_left foo b = function
    | [] -> b
    | t::q -> fold_left foo (foo b t) q;;

val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list
                    -> 'a = <fun>
```

Avantages et inconvénients de `List.fold_*`

Parfois très utile pour traiter succintement des listes

Cependant :

- pas toujours la solution la plus simple
- peut être délicat à écrire correctement
- peut être *très* difficile à lire

Avantages et inconvénients de `List.fold_*`

Parfois très utile pour traiter succintement des listes

Cependant :

- pas toujours la solution la plus simple
- peut être délicat à écrire correctement
- peut être *très* difficile à lire

Détaillez votre démarche (avec un dessin !)

Avantages et inconvénients de `List.fold_*`

Parfois très utile pour traiter succintement des listes

Cependant :

- pas toujours la solution la plus simple
- peut être délicat à écrire correctement
- peut être *très* difficile à lire

Détaillez votre démarche (avec un dessin !)

Ne vous forcez jamais à trouver une solution de ce type

Avantages et inconvénients de `List.fold_*`

Parfois très utile pour traiter succintement des listes

Cependant :

- pas toujours la solution la plus simple
- peut être délicat à écrire correctement
- peut être *très* difficile à lire

Détaillez votre démarche (avec un dessin !)

Ne vous forcez jamais à trouver une solution de ce type

Il existe *toujours* d'autres solutions

Un exemple défendable : le tri insertion


```
# let tri_insertion lst =
    let rec insertion lst elem = match lst with
      | [] -> [nw]
      | t::q when elem>t -> t::insertion q elem
      | lst -> elem::lst

    in List.fold_left insertion [] lst;;

val tri_insertion : 'a list -> 'a list = <fun>
```

Aplatir une liste

On peut écrire `aplatir` avec une fonction récursive :

```
# let rec aplatir = function
    | [] -> []
    | lst::[] -> lst
    | []::q -> aplatir q
    | (t::q)::q2 -> t::aplatir (q::q2);;

val supprime_doublons : 'a list list -> 'a list = <fun>
```

Supprimer les doublons

On peut écrire supprime_doublons avec une fonction récursive :

```
# let rec supprime_doublons = function
    | [] -> []
    | t::q when List.mem t q -> supprime_doublons q
    | t::q -> t :: supprime_doublons q;;

val supprime_doublons : 'a list -> 'a list = <fun>
```

Retourner une liste

On peut également écrire `retourne` avec une fonction récursive :

```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```


`retourne [ 1; 2; 3 ]`

Retourner une liste

On peut également écrire retourne avec une fonction récursive :

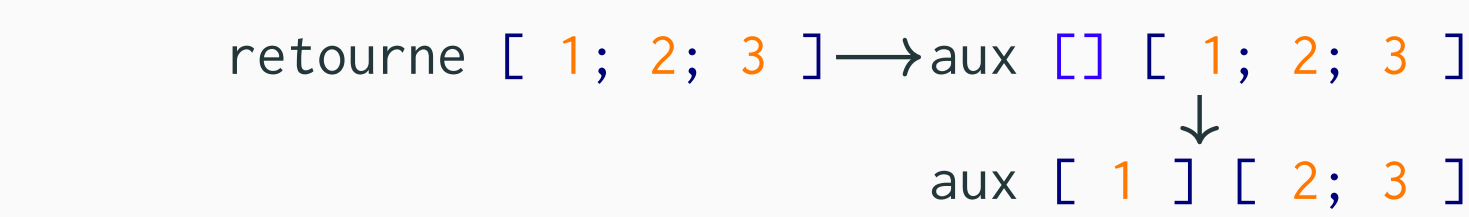
```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```


retourne [1; 2; 3] ⟶ aux [] [1; 2; 3]

Retourner une liste

On peut également écrire `retourne` avec une fonction récursive :

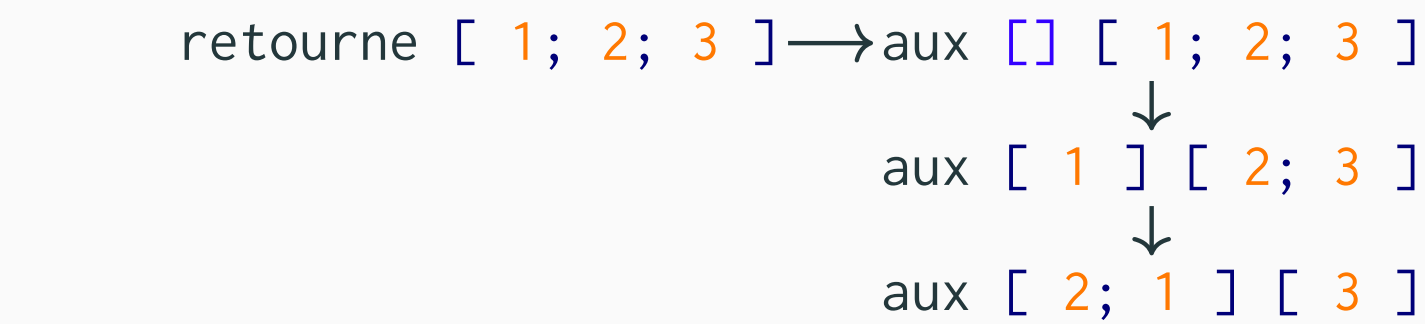
```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```



Retourner une liste

On peut également écrire `retourne` avec une fonction récursive :

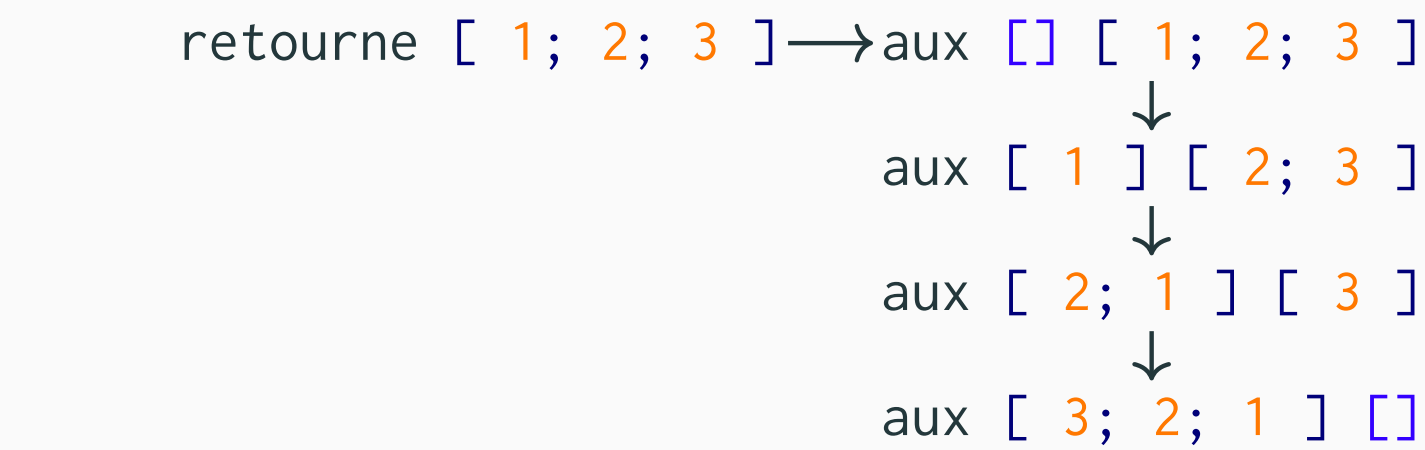
```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```



Retourner une liste

On peut également écrire `retourne` avec une fonction récursive :

```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```



Retourner une liste

On peut également écrire retourne avec une fonction récursive :

```
let retourne lst =
  let rec aux res = function
    | [] -> res
    | t::q -> aux (t::res) q
  in aux [] lst;;
```