3.10. Listes

3.10.1. Principes généraux

Le langage OCaml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes homogènes, sans avoir à les redéfinir.

Une *liste homogène* est une suite finie d'éléments d'un même type.

Les listes sont notées avec des crochets et des points-virgules. Ainsi,

```
[e1; e2; ... ; en]
```

est une liste contenant, de gauche à droite, les valeurs des expressions `e1`, `e2`, ..., `en`.

La *liste vide* est notée `[]`.

Le type `'a list` est un **type polymorphe** et `[]` est une **valeur polymorphe**.

Exemples

```
# [2; 4; 5 + 3; 16];;
- : int list = [2; 4; 8; 16]
```

```
# [];;
- : 'a list = []
```

L'*opérateur de construction* `::` est un opérateur infixe d'arité deux.

L'expression `e :: lst` est équivalente à l'appel `List.cons e lst`, où `List.cons` est la fonction de type

`'a -> 'a list -> 'a list`

telle que si `e` est un élément et `lst` est une liste, alors `List.cons e lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

Exemples

```
# 2 :: [1; 2; 3];;
- : int list = [2; 1; 2; 3]
```

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

```
# [1; 2] |> List.cons 0 |> List.cons 3;;
- : int list = [3; 0; 1; 2]
```

L'opérateur `::` est **associatif de droite à gauche**.

Exemple

L'expression

`1 :: 2 :: 3 :: []`

désigne l'expression totalement parenthésée

`(1 :: (2 :: (3 :: [])))`

L'opérateur de construction est également un **opérateur de déconstruction** lorsque nous nous en servons dans un filtrage de motifs.

Exemple

```
let tete lst =
    match lst with
        |[] -> failwith "liste vide"
        |e :: _ -> e
```

```
let tete_opt lst =
    match lst with
        |[] -> None
        |e :: _ -> Some e
```

Ces filtrages déconstruisent la liste en argument pour accéder à son 1er élément.
La fonction `failwith` est de type `string -> 'a`. Elle provoque une erreur volontaire.

Exemple

```
let rec un_sur_deux lst =
    match lst with
        |[] -> []
        |[e] -> [e]
        |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste)
```

Ceci renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : il s'agit de **fonctions polymorphes**.

3.10.2. Opérations sur les listes

La bibliothèque standard d'OCaml (`Stdlib`) propose à travers le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	Longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombres d'éléments dans les listes impliquées.

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **créer** une liste selon une spécification ;
2. **transformer** les éléments d'une liste ;
3. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
4. **tester** si tous les (resp. au moins un) élément.s d'une liste vérifie.nt une propriété ;
5. **combiner** les éléments d'une liste pour calculer une valeur ;
6. **permuter** les éléments d'une liste.

Exemples

1. [Création] construire la liste des caractères qui composent une chaîne de caractères, construire la liste des nombres de Fibonacci inférieurs à 512 ;
2. [transformation] multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;
3. [sélection] obtenir la sous-liste des nombres pairs d'une liste d'entiers ;
4. [test] tester si toutes les chaînes de caractères contenues dans une liste commencent par le caractère `'a'`, tester la présence de l'élément `7` dans une liste d'entiers ;
5. [combinaison] calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;
6. [permutation] tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de fonctions d'ordre supérieur.

Pour **créer une liste**, nous spécifions sa longueur n désirée et ses éléments en fonction de leur position i dans la liste, via une fonction f. La valeur de élément d'indice i est $f(i)$.

Une telle fonction f est ainsi de type `int -> 'a`. La **fonction de création** `initialiser` est de type

```
int -> (int -> 'a) -> 'a list
```

et sa définition est

```
let initialiser n f =
    let rec aux i acc =
        if i < 0 then
            acc
        else
            aux (i - 1) (f i :: acc)
    in
    aux (n - 1) []
```

Exemples

```
# initialiser 8 Fun.id;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7]
```

```
# initialiser 6 ((<=) 2);;
- : bool list = [false; false; true; true; true; true]
```

Nous avons réimplanté ici la fonction `List.init`.

Pour calculer l'**image miroir** d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution **récursive terminale** :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> aux reste (e :: acc)
    in
aux lst []
```

Exemples

```
# miroir ['a'; 'b'; 'c'];;
- : char list = ['c'; 'b'; 'a']
```

```
# miroir [1; 1; 2; 2; 2; 1];;
- : int list = [1; 2; 2; 2; 1; 1]
```

Nous avons réimplanté ici la fonction `List.rev`.

Une bonne manière de spécifier la manière de **transformer** les éléments d'une liste consiste à donner une fonction f telle que chaque élément x de la liste est transformé en l'élément $f(x)$.

Une telle fonction f est ainsi de type `'a -> 'b`. La **fonction de transformation** `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer f lst =
    match lst with
        |[] -> []
        |e :: reste -> (f e) :: (transformer f reste)
```

Exemples

```
# transformer (fun x -> x * 3) [1; 2; 3; 4];;
- : int list = [3; 6; 9; 12]
```

```
# transformer int_of_char ['a'; 'b'; 'c'; '1'];;
- : int list = [97; 98; 99; 49]
```

Nous avons réimplanté ici la fonction `List.map`.

Une bonne manière d'**extraire une sous-liste** d'une liste consiste à donner une fonction f qui est telle que $f(x)$ est vrai si l'élément x est à conserver. La fonction f est dans ce cas un *prédicat*.

Une telle fonction f est ainsi de type `'a -> bool`. La **fonction de sélection** `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner f lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = selectionner f reste in
            if f e then e :: suite else suite
```

Exemple

```
# selectionner (fun x -> x mod 2 = 0) [13; 8; 9; 8; 6; 15; 2];;
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `List.filter`.

Une bonne manière de **tester si tous les éléments** d'une liste vérifient une propriété donnée consiste à représenter cette propriété par un prédicat f qui est tel que $f(x)$ est vrai si x vérifie la propriété.

La **fonction de test universel** `tester_univ` est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ f lst =
    match lst with
        |[] -> true
        |x :: _ when not (f x) -> false
        |_ :: reste -> tester_univ f reste
```

Exemple

```
# tester_univ (fun u -> u.[0] = 'a') ["a"; "aba"; "abacaba"; "abacabadabacaba"];;
- : bool = true
```

Nous avons réimplanté ici la fonction `List.for_all`.

Une bonne manière de **tester s'il existe un élément** d'une liste qui vérifie une propriété donnée consiste à représenter cette propriété par un prédicat f qui est tel que $f(x)$ est vrai si x vérifie la propriété.

La **fonction de test existentiel** `tester_exist` est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist f lst =
    match lst with
        |[] -> false
        |x :: _ when f x -> true
        |_ :: reste -> tester_exist f reste
```

Exemple

```
# tester_exist ((=) 7) [0; 1; 1; 2; 3; 5; 8];;
- : bool = false
```

Nous avons réimplanté ici la fonction `List.exists`.

La fonction `List.mem` peut s'implanter par

```
let mem x lst = lst |> tester_exist ((=) x)
```

La problématique ici est la suivante. Nous disposons

1. d'un élément e ;
2. d'une suite d'éléments $(e_1, e_2, \ldots, e_n)$
3. d'une opération binaire $\bullet$;

et nous souhaitons calculer

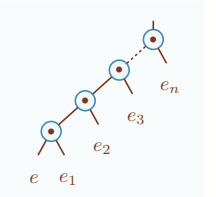
$$(\cdots((e \bullet e_1) \bullet e_2) \bullet \cdots) \bullet e_n.$$

Exemples

Ce problème admet de **nombreuses instances**, à première vue de nature différente :

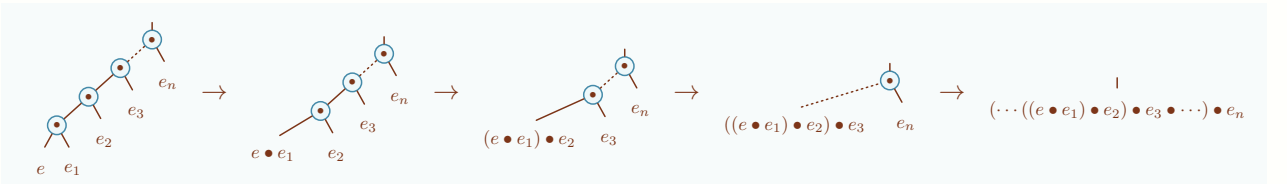
- lorsque $e = 0$, $(e_1, e_2, \ldots, e_n)$ est une suite d'entiers et $\bullet$ est l'opération d'**addition** des entiers, ceci calcule la **somme** des éléments de la suite ;
- lorsque $e = \epsilon$ (mot vide), $(e_1, e_2, \ldots, e_n)$ est une suite de chaînes de caractères et $\bullet$ est l'opération de **concaténation**, ceci calcule la **concaténation générale** des chaînes de caractères de la suite ;
- lorsque $e = e_1$, $(e_1, e_2, \ldots, e_n)$ est une suite de nombres telle que $n \geqslant 1$ et $\bullet$ est l'opération « $\max$ », ceci calcule la **plus grande valeur** de la suite.

En d'autres termes, étant donnés l'élément e, la suite d'éléments $(e_1, e_2, \ldots, e_n)$ et l'opération binaire $\bullet$, nous souhaitons **évaluer** l'arbre syntaxique



L'élément e est l'élément initial pour le calcul. Il s'agit de la *graine*.

Ce calcul s'exprime **récursivement** en notant que la valeur recherchée est e lorsque $n = 0$ et sinon, elle est celle calculée dans le cas où la graine est $e \bullet e_1$ et la suite est $(e_2, \ldots, e_n)$:



Ceci s'appelle le *pliage à gauche*.

La suite d'éléments $(e_1, \ldots, e_n)$ est représentée par une liste `lst`, l'opération $\bullet$, par une fonction `f` de type `'a -> 'a -> 'a` et la graine `e`, par un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche f e lst =
    match lst with
        |[] -> e
        |x :: reste -> pliage_gauche f (f e x) reste
```

Exemples

```
# pliage_gauche (+) 0 [0; 2; 1; 5; 2; -2; 3];;
- : int = 11
```

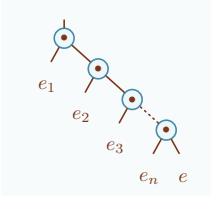
```
# pliage_gauche (^) "" ["mi"; "la"; "re"; "sol"; "do"; "fa"];;
- : string = "milaresoldofa"
```

```
# let lst = [0; 2; 1; 5; 2; -2] in
pliage_gauche max (List.hd lst) (List.tl lst);;
- : int = 5
```

Il est possible de faire essentiellement la même chose, mais pour calculer plutôt

$$e_1 \bullet (e_2 \bullet (e_3 \bullet (\cdots (e_n \bullet e) \cdots)))$$

Ceci se fait en évaluant récursivement l'arbre syntaxique



Il s'agit de l'opération de *pliage à droite* qui est de type

```
('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

et dont la définition est

```
let rec pliage_droite f lst e =
    match lst with
        |[] -> e
        |x :: reste -> f x (pliage_droite f reste e)
```

Ces deux opérations de pliage existent sous les noms respectifs de `List.fold_left` et `List.fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
```

D'un point de vue sémantique, lorsque l'opération $\bullet$ est **associative** et que la graine e **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Il y a une différence d'efficacité : le pliage à gauche est **récursif terminal** alors que le pliage à droite ne l'est pas. En effet, dans le pliage à gauche, **la graine est l'accumulateur**.

Cela dit, le pliage à droite reste intéressant (voire indispensable) lorsque les opérations doivent être réalisées dans un ordre précis. Ceci est illustré dans les prochains exemples.

Les pliages sont des opérations très puissantes : ils permettent d'émuler beaucoup de fonctions d'ordre supérieur sur les listes :

Émulation de `List.rev` :

```
let rev lst =
    lst |> List.fold_left (fun res x -> x :: res) []
(* Plus esthétique :      (Fun.flip List.cons) *)
```

Émulation de `List.map` :

```
let map f lst =
    List.fold_right (fun x res -> f x :: res) lst []
```

Émulation de `List.filter` :

```
let filter f lst =
    List.fold_right (fun x res -> if f x then x :: res else res) lst []
```

Émulation de `List.for_all` :

```
let for_all f lst =
    lst |> List.fold_left (fun res x -> f x && res) true
```

Émulation de `List.exists` :

```
let exists f lst =
    lst |> List.fold_left (fun res x -> f x || res) false
```

Les deux utilisations de `List.fold_right` sont importantes ici pour construire une liste dont les éléments apparaissent dans le bon ordre.

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
`init`	`int -> (int -> 'a) -> 'a list`
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Il est important en pratique de veiller à ne pas réimplanter (des variantes de) ces fonctions.

Il existe aussi la fonction `sort` de type

```
('a -> 'a -> int) -> 'a list -> 'a list
```

renvoyant une version triée d'une liste au moyen d'une fonction de comparaison (1er paramètre).

3.10.3. Exemples de manipulation de listes

Les fonctions opérant sur les listes qui acceptent une **liste en dernier argument** sont adaptées pour être utilisées conjointement à l'**opérateur d'application inversée** `|>`.

Exemples

- Si `lst` est une liste de couples,

```
lst |> List.map (fun (x, y) -> (y, x))
```

  est la liste obtenue en transposant les couples de `lst`.

- Si `lst` est une liste d'entiers,

```
lst |> List.map (fun x -> x * (-2)) |> List.fold_left (+) 0
```

  est la somme des entiers de `lst` multipliés au préalable par `-2`.

- Si `lst` est une liste de listes,

```
lst |> List.map List.length |> List.exists (fun x -> x mod 2 = 0)
```

  teste s'il existe une liste élément de `lst` qui est de longueur paire.

- Si `lst` est une liste de chaînes de caractères,

```
lst |> List.filter (fun u -> u >= "ab") |> List.map String.lowercase_ascii |> List.fold_left (^) ""
```

  est la concaténation des éléments de `lst` supérieurs à `"ab"` puis convertis en minuscules.

Par manipulation de listes, il est possible de proposer la version suivante du calcul de la factorielle d'un entier positif :

```
let fact n =
    List.init n succ |> List.fold_left ( * ) 1
```

```
# List.init 8 fact;;
- : int list = [1; 1; 2; 6; 24; 120; 720; 5040]
```

Lors d'un appel `fact n`, elle procède ainsi :

1. elle commence par construire la liste `[1; 2; ...; n]` via l'appel à `List.init`.

   Notons l'utilisation de la fonction `succ` qui fait qu'en position `i`, la liste construite contient la valeur `i + 1` ;

2. ensuite, elle plie cette liste par multiplication via l'appel à `List.fold_left`.

En suivant des idées similaires à celles de l'exemple précédent, il est possible de proposer la version suivante du calcul du nombre de Fibonacci d'un entier positif :

```
let fibonacci n =
    let next (a, b) =
        (b, a + b)
    in
    List.init n (Fun.const ())
    |> List.fold_left (fun res _ -> next res) (0, 1)
    |> fst
```

```
# List.init 12 fibonacci;;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Lors d'un appel `fibonacci n`, elle procède ainsi :

1. elle commence par construire la liste support `[(); ...; ()]` de longueur `n` via l'appel à `List.init` ;
2. ensuite, elle plie cette liste en construisant un couple, initialement `(0, 1)`, en le mettant à jour via la fonction `next`.

   Cette fonction `next` contient le mécanisme de calcul de Fibonacci : elle transforme un couple (a_{n-1}, a_n) en le couple $(a_n, a_n + a_{n-1}) = (a_n, a_{n+1})$, où a_m est le m^{e} nombre de Fibonacci.