PG104 - Programmation Fonctionnelle

D. Renault

ENSEIRB-Matmeca

3 février 2025, v.1.2.5

https://www.labri.fr/perso/renault/working/teaching/functional/functional.php

Structures de données fonctionnelles

Principe général

Écrire du code en minimisant les dépendances pour mieux les contrôler.

- Un cas d'application central de la gestion des dépendances : les structures de données.
- Peut-on profiter de la pureté en manipulant des données ?
  Peut-on programmer avec des données non mutables ?

Définition (Structure de données fonctionnelle)

Une structure de données est (purement) **fonctionnelle** si elle peut être implémentée de manière pure. Ces structures sont forcément immutables.

Idée : mettre à profit la récursivité au sein même des structures de données.

Types de données inductifs

Définition (Type de données inductif)

Un type de données $\mathcal{T}$ est dit **inductif** si sa structure est récursive.
Certains de ses composants sont du même type que $\mathcal{T}$.

Exemple : la liste chaînée

Une liste chaînée est un type de données possédant une tête et une queue.
Soit ces 2 composants sont `null`, soit la queue est **aussi** une liste chaînée.

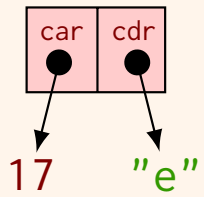
- Nommés aussi types de données algébriques, types sommes ou variants.

```
type List = { head: any, tail: List }          // Typescript definition
```

- Deux exemples fondamentaux de types inductifs : les **listes** et les **arbres**.
- Un type inductif n'est pas forcément immutable / fonctionnel.

Définition (Paire pointée)

Une **paire pointée** (appelée aussi *cons* ou *construct*) est une structure de donnée contenant deux références appelées traditionnellement `car` et `cdr`.



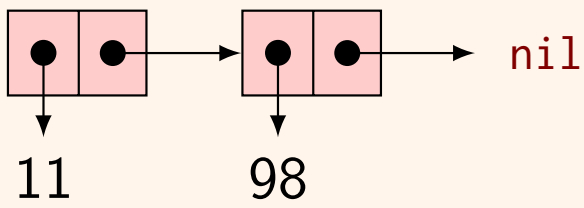

```
{ car: 17, cdr: "e" }
```

```
// constructor
function cons(aCar, aCdr) { return { car: aCar, cdr: aCdr }; }
// accessors
function car(aCons)       { return aCons['car']; }
function cdr(aCons)       { return aCons['cdr']; }
```

Définition (Liste)

Une **liste** (*list*) est une structure de données définie de manière inductive de la façon suivante :

- la liste vide `nil` est une liste ;
- si `e` est une valeur et `l` est une liste, alors `cons(e, l)` est une liste.



`cons(11, cons(98, nil))`

```
// constructors
const nil = {};
// accessors
function head(l)    { return car(l); }
function tail(l)    { return cdr(l); }
// predicates
function isEmpty(l) { return l === nil;}
```

Comparaison liste / tableau

Les types “liste” et “tableau” partagent de fortes ressemblances, mais :

- Leurs propriétés algorithmiques sont différentes :

Complexités en moyenne	Accès	Mise à jour	Recherche	Insertion (après recherche)	Suppression (après recherche)
Tableau	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Liste	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- Leur gestion de la mémoire diffère (contiguë / par cellules) ce qui peut aussi influencer l’efficacité.
- La possibilité de décomposer les listes **inductivement** facilite leur association avec des techniques comme la pureté ou l’immutabilité.

Exemple d'algorithme sur les listes : sum

Considérons le problème consistant à sommer une liste d'entiers :

```
function listSum(l) {                          // (recursive version)
  if (isEmpty(l))                              // • if the list is empty
    return 0;                                  //    a default value is returned
  else                                         // • if the list is not empty
    return head(l) + listSum(tail(l));         //    recursively handle head and tail
}
```

Les algorithmes manipulant des listes ont tendance à tous se ressembler.

Influence de la structure inductive

Structure inductive des listes $\Rightarrow$ Structure inductive des algorithmes

Exemple d'algorithme sur les listes : sum

Considérons le problème consistant à sommer une liste d'entiers :

```
function listSum(l, cpt) {                    // (tail-recursive version)
  if (isEmpty(l))                             // • if the list is empty
    return cpt;                               //    a default value is returned
  else                                        // • if the list is not empty
    return listSum(tail(l),cpt+head(l)); //    recursively handle head and tail
}
```

Les algorithmes manipulant des listes ont tendance à tous se ressembler.

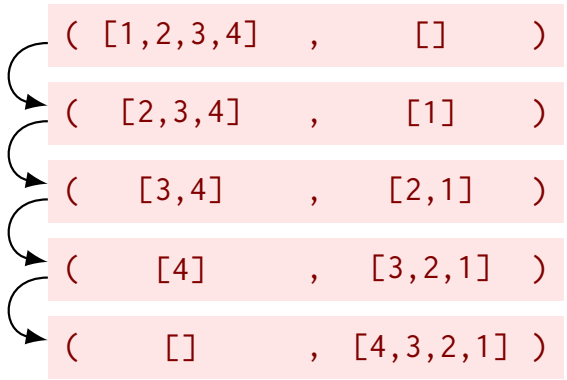
Influence de la structure inductive

Structure inductive des listes $\Rightarrow$ Structure inductive des algorithmes

Exemple d'algorithme sur les listes : reverse

Considérons le problème consistant à **renverser une liste** de manière **récursive terminale** :

```
function reverse(list, res) {
  if (isEmpty(list))
    return res;
  else
    return reverse(tail(list),
                   cons(head(list), res));
}
```



Principe

La récursivité terminale sur les listes se traduit en un jeu de réécriture des paramètres, à la manière des tours de Hanoï.

La bibliothèque `list`

Il existe plusieurs bibliothèques `npm` orientée sur la gestion des listes.
Exemple notable : la bibliothèque `list` (`https://github.com/funkia/list`).

```
import * as L from "list";
```

```
function countChars(str, c) {                    // -- List Functions --
    if (L.isEmpty(str))                          // L.isEmpty
        return 0;                                //
    else if (L.head(str) === c)                  // L.head
        return 1 + countChars(L.tail(str), c);   // L.tail
    else                                         //
        return countChars(L.tail(str), c);       // L.tail
}
```

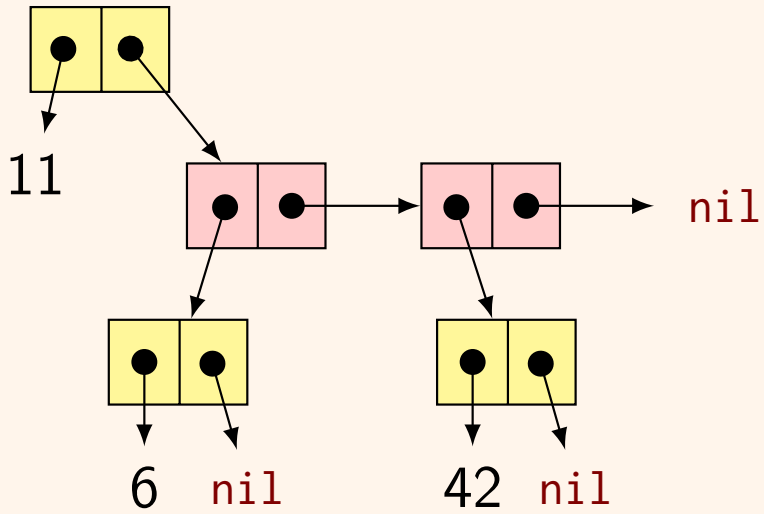
Parmi les caractéristiques mises en avant :

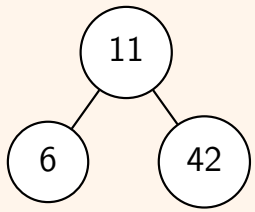
- les listes sont immutables, et donc adaptées à la programmation pure ;
- les listes sont immutables, permettant des optimisations comparables à ce que l'on peut obtenir avec des tableaux.

Définition (Arbre)

Un **arbre** (*tree*) est une structure de données définie de manière inductive, et possédant deux références :

- une valeur `val`,
- et une liste `children` ne contenant que des arbres.






```
node(11,
     cons(node(6, nil),
          cons(node(42, nil),
               nil)))
```

```
function node(aVal, somChld) { return { val: aVal, children: somChld }; }
function val(aTree)          { return aTree['val']; }
function children(aTree)     { return aTree['children']; }
```

Les arbres sont un exemple de **composition** de types de données inductifs.

- Un arbre est une **paire pointée** dont l'un des éléments est une **liste**.
- Les fonctions sur les arbres utilisent donc naturellement celles sur les paires pointées et celles sur les listes.

Principe (conception de types)

Composer les types simples en des types plus complexes.

. . . un peu comme les expressions.

Exemple d'algorithme sur les arbres : size

Considérons le problème de calculer le nombre de sommets dans un arbre :

```
function treeSize(t) {                  // t  is simply a tree
    function listTreeSize(tl) {         // tl is a list of trees
        if (isEmpty(tl))
            return 0;
        else
            return treeSize(head(tl)) + listTreeSize(tail(tl));
    }
    return 1 + listTreeSize(children(t));
}
```

- Souligne la composition entre les fonctions sur les arbres et les listes.
- Un problème se pose si chaque fonction sur les arbres demande à écrire une fonction sur les listes.

Structures inductives et récursivité

Le lien naturel entre structures inductives et récursivité aide à la conception d'algorithmes sous forme de règles de réécritures.

Exemple : calcul de la longueur d'une liste

Selon la forme de `aList`, sa longueur vaut,

- si `aList` est `nil`,
  `length(nil)` $\longrightarrow$ `0`
- si `aList` est de la forme `cons(aHd, aTl)`,
  `length(cons(aHd, aTl))` $\longrightarrow$ `1 + length(aTl)`

en Javascript :

```
function listLength(aList) {
    if (isEmpty(aList))
        return 0;
    else // aList has a tail
        return 1 + listLength(tail(aList));
}
```

Structures inductives et récursivité

Le lien naturel entre structures inductives et récursivité aide à la conception d'algorithmes sous forme de règles de réécritures.

Exemple : calcul de la longueur d'une liste

Selon la forme de `aList`, sa longueur vaut,

- si `aList` est `nil`,
  `length(nil)` $\longrightarrow$ `0`
- si `aList` est de la forme `cons(aHd, aTl)`,
  `length(cons(aHd, aTl))` $\longrightarrow$ `1 + length(aTl)`

en Javascript (`switch`) :

```
function listLength(aList) {
    switch(true) {
    case isEmpty(aList):
        return 0;
    default: // aList has a tail
        return 1 + listLength(tail(aList));
} }
```

Structures inductives et récursivité

Le lien naturel entre structures inductives et récursivité aide à la conception d'algorithmes sous forme de règles de réécritures.

Exemple : calcul de la longueur d'une liste

Selon la forme de `aList`, sa longueur vaut,

- si `aList` est `nil`,
  `length(nil)` $\longrightarrow$ `0`
- si `aList` est de la forme `cons(aHd, aTl)`,
  `length(cons(aHd, aTl))` $\longrightarrow$ `1 + length(aTl)`

en OCaml :

```
let rec length aList = match aList with
    | nil      -> 0
    | aHd::aTl -> 1 + length(aTl);
;;
```

Le code en OCaml est une forme de **reconnaissance de motifs** (*pattern-matching*).

Structures inductives et récursivité

Le lien naturel entre structures inductives et récursivité aide à la conception d'algorithmes sous forme de règles de réécritures.

Exemple : calcul de la longueur d'une liste

Selon la forme de `aList`, sa longueur vaut,

- si `aList` est `nil`,
  `length(nil)` → `0`
- si `aList` est de la forme `cons(aHd, aTl)`,
  `length(cons(aHd, aTl))` → `1 + length(aTl)`

en OCaml :

```
let rec length aList = match aList with
    | nil      → 0
    | aHd::aTl → 1 + length(aTl);
;;
```

Le code en OCaml est une forme de **reconnaissance de motifs** (*pattern-matching*). Le langage Javascript ne possède pas une syntaxe aussi avancée, même s'il existe une proposition d'extension (actuellement en pause).

La reconnaissance de motifs et les algorithmes

La reconnaissance de motifs est particulièrement bien adaptée pour décrire des algorithmes qui agissent par transformations / réécritures.

Exemple : équilibrer les arbres rouge-noir

Après insertion dans un arbre rouge-noir, il faut rééquilibrer [Oka99] :

```
let balance colour left root right =
  match colour, left, root, right with
 | Black, T (Red, T (Red, a, x, b), y, c), z, d   (* rotate in these 4 cases *)
 | Black, T (Red, a, x, T (Red, b, y, c)), z, d
 | Black, a, x, T (Red, T (Red, b, y, c), z, d)
 | Black, a, x, T (Red, b, y, T (Red, c, z, d)) → T (Red, T (Black, a, x, b), y, T (Black, c, z, d))
 | _                                            → T (colour, left, root, right) (* otherwise unchanged *)
```

Code tiré de github/pewulfman, algorithme décrit sur Wikipedia.

En bonus, ces algorithmes sont automatiquement **récursifs terminaux**.

La manipulation de types de données immutables pose un gros problème :

Comment gérer la multiplication des instances ?

- La gestion de la mémoire est automatique, il faut lui faire confiance. Cf. optimisations complexes du ramasse-miettes de V8, Orinoco.
- L'immutabilité permet de mitiger le nombre d'instances en mémoire $\Rightarrow$ la **persistance**.

La persistance

Définition (Persistance)

Propriété logicielle selon laquelle une structure de données persiste en mémoire à ses traitements. Les opérations sur ces structures doivent recréer de nouvelles instances indépendantes des anciennes à chaque opération.

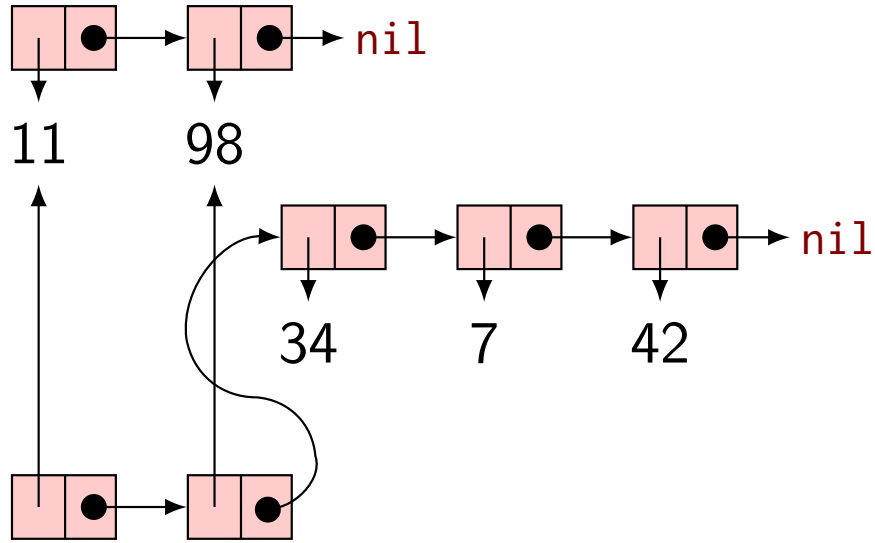
Fait

Une structure de données pure ou immutable est forcément persistante.

- Avantages : les instances sont indépendantes (on peut faire des calculs indépendants dessus), et elles persistent (on peut revenir dans le passé)
- Inconvénients : les instances ont tendance à s'accumuler au fur et à mesure si on ne gère pas la libération (coût mémoire).

Comme les instances sont indépendantes, il est possible de les **réutiliser**.

```
function append(l1, l2) {
  if (isEmpty(l1))
    return l2;
  else
    return cons(head(l1),
                append(tail(l1),l2));
}
append([11,98], [34,7,42]);
```



- La réutilisation permet de profiter des données accessibles.
- Le ramasse-miettes se charge de récupérer la place prise par les données inaccessibles.

Conclusion sur la persistance

La multiplication des instances peut être mitigée à l'aide de la persistance.

Opérateurs d'ordre supérieur

Revenons sur l'algorithme suivant censé représenter une boucle générique :

```
function loop(block, init, times) {    // block is a 1-parameter function
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

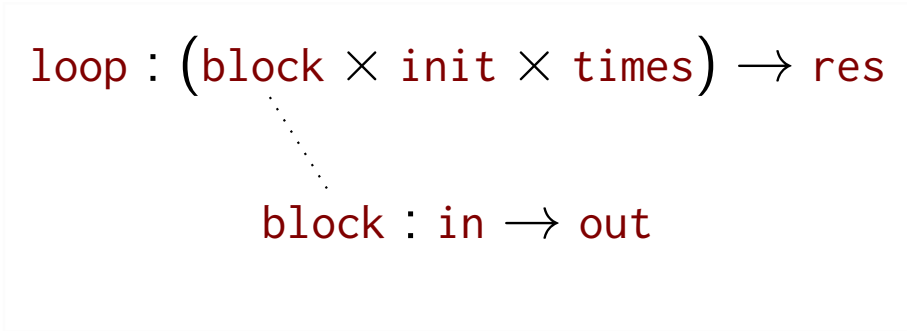
- `loop` est une fonction qui prend en paramètre une fonction `block`.
  Il s'agit d'une **fonction d'ordre supérieur**.
- Décrit un algorithme complexe à partir d'un algorithme plus simple.

Quels genres d'algorithmes de cette forme existent sur les tableaux / listes ?

Ordre supérieur et types

Comment appréhender des fonctions prenant des fonctions en paramètre ?
Un début de réponse peut être apporté en considérant les types des objets :

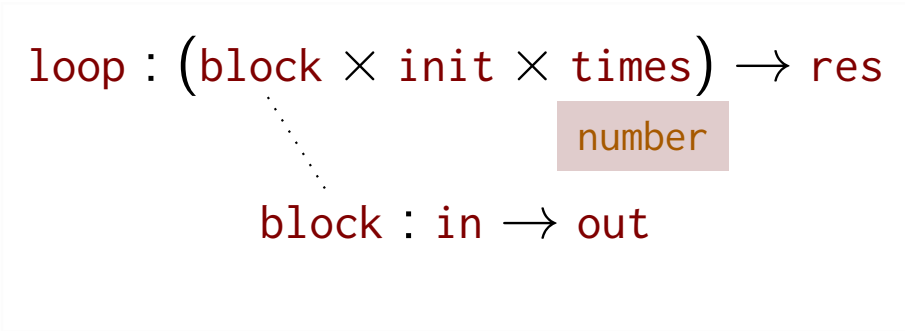
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



Ordre supérieur et types

Comment appréhender des fonctions prenant des fonctions en paramètre ?
Un début de réponse peut être apporté en considérant les types des objets :

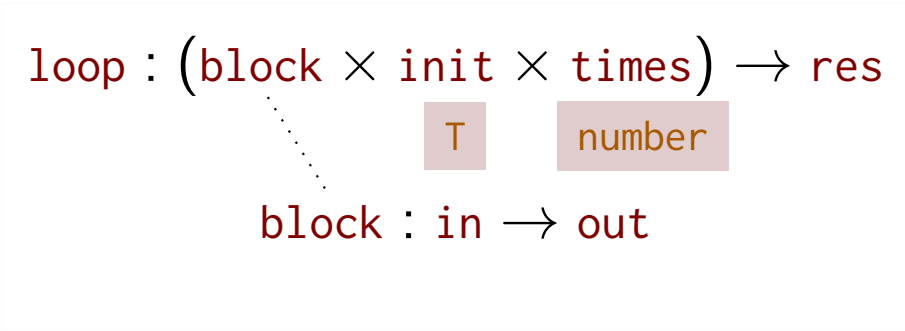
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



Ordre supérieur et types

Comment appréhender des fonctions prenant des fonctions en paramètre ?
Un début de réponse peut être apporté en considérant les types des objets :

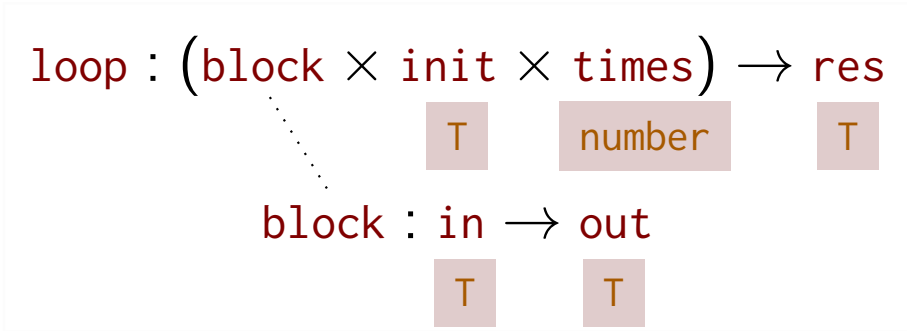
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



Ordre supérieur et types

Comment appréhender des fonctions prenant des fonctions en paramètre ?
Un début de réponse peut être apporté en considérant les types des objets :

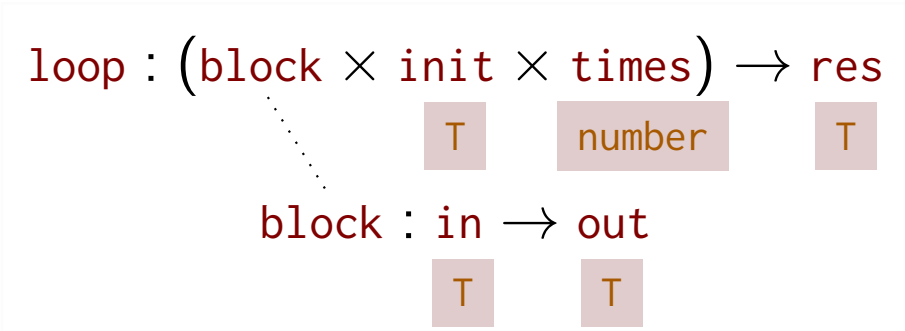
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



Ordre supérieur et types

Comment appréhender des fonctions prenant des fonctions en paramètre ?
Un début de réponse peut être apporté en considérant les types des objets :

```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



- Le type de la fonction `loop` s'écrit : `loop : ( (T → T), T, number ) → T`

```
function loop<T>(block: (T) ⇒ T, init: T, times: number): T
```

- `T` peut être n'importe quel type, ce qui rend la fonction **générique**.
  Cette généricité émerge naturellement avec les fonctions d'ordre supérieur.

Ordre supérieur : l'itérateur forEach

Le `forEach` : réaliser une boucle sur les élements **d'un tableau** :

```
function forEach<T>(block: (T) ⇒ void, arr: T[]) : void {
    for (let i = 0; i < arr.length; i++) {
        block(arr[i]);
    }
}

forEach((x) ⇒ { console.log(x); }, [1,2,3,4]); // Log : 1, 2, 3, 4
```

```
import * as _ from 'underscore';
```

```
_.each([1,2,3,4], (x) ⇒ { console.log(x); });   // function-version (underscore)
[1,2,3,4].forEach((x) ⇒ { console.log(x); });   // method-version (stdlib)
```

Ordre supérieur : l'itérateur forEach

Le `forEach` : réaliser une boucle sur les élements **d'une liste** :

```
function forEach<T>(block: (T) ⇒ void, lst: List[T]) : void {
    if (isEmpty(lst))
        return;
    else {
        block(head(lst)); forEach(block, tail(lst));
    }
}
```

```
let list_1234 = cons(1, cons(2, cons(3, cons(4, nil))));
forEach((x) ⇒ { console.log(x); }, list_1234); // Log : 1, 2, 3, 4
```

Ordre supérieur : la transformation map

Le `map` : appliquer une transformation à chaque élément d'une liste / tableau

```
function map<T,U>(block: (T) ⇒ U, arr: T[]): U[] {
    const brr = [];
    for (let i = 0; i < arr.length; i++)
        brr[i] = block(arr[i]);
    return brr;
}

map((x) ⇒ x+2, [1,2,3,4]); // → [3,4,5,6]
```

```
import * as _ from 'underscore';
```

```
_.map([1,2,3,4], (x) ⇒ x+2); // function-version (underscore)
   [1,2,3,4].map((x) ⇒ x+2); // method-version (stdlib)
```

Ordre supérieur : la sélection filter

Le `filter` : sélectionner des éléments dans une liste / tableau

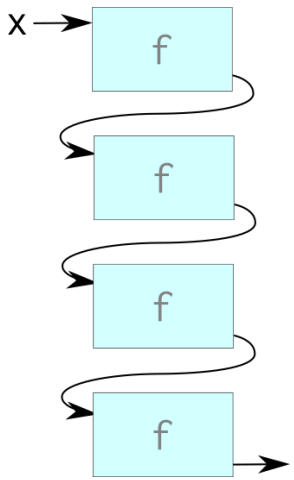
```
function filter<T>(block: (T) ⇒ boolean, arr: T[]) : T[] {
    const brr = [];
    for (let i = 0, j = 0; i < arr.length; i++) {
        if (block(arr[i]))
            brr[j++] = arr[i];
    }
    return brr;
}

filter((x) ⇒ x%2 === 0, [1,2,3,4,5,6]); // → [2,4,6]
```

```
import * as _ from 'underscore';
```

```
_.filter([1,2,3,4,5,6], (x) ⇒ x%2 === 0);   // function-version (underscore)
  [1,2,3,4,5,6].filter((x) ⇒ x%2 === 0);   // method-version (stdlib)
```

Ordre supérieur : les pliages (1/3)

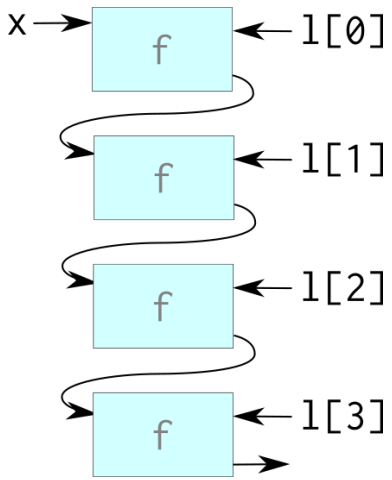
Quelle est la version fonctionnelle pure d'un **parcours sur une liste** `l` ?
Pour rappel, nous avions établi le schéma suivant pour une simple boucle :




```
function recur(block, init, times) {
  if (times === 0)
    return init;
  else {
    const ninit = block(init);
    return recur(block, ninit, times-1);
} }
```

Ordre supérieur : les pliages (1/3)

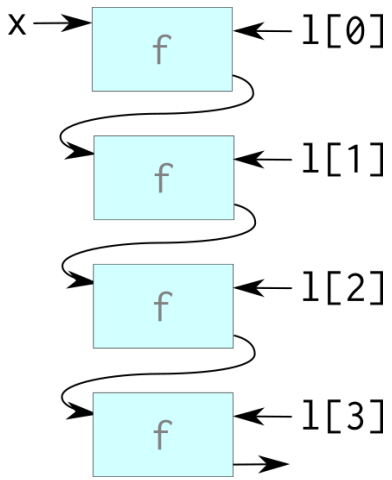
Quelle est la version fonctionnelle pure d'un **parcours sur une liste** `l` ?
Une idée simple consiste à passer les éléments de la liste `l` dans la boucle :




```
function recur(block, init, times) {
  if (times === 0)
    return init;
  else {
    const ninit = block(init);
    return recur(block, ninit, times-1);
} }
```

Ordre supérieur : les pliages (1/3)

Quelle est la version fonctionnelle pure d'un **parcours sur une liste** `l` ?
Une idée simple consiste à passer les éléments de la liste `l` dans la boucle :




```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Cette fonction s'appelle un **pliage** (*fold* ou *reduce*).

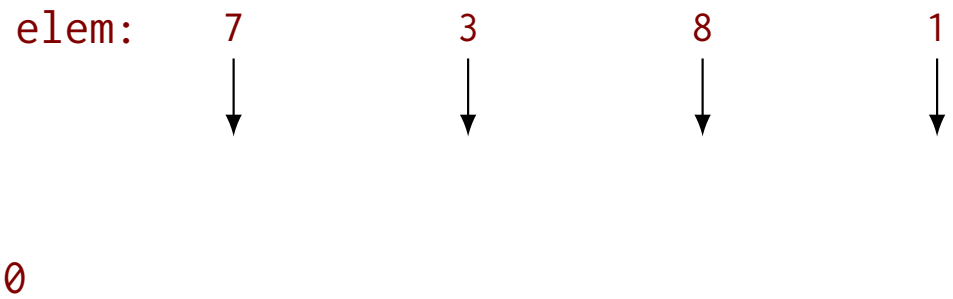
Ordre supérieur : les pliages (2/3)

Considérons l'exemple suivant d'exécution de `reduce` :

- *init* = `0`
- *list* = `[7,3,8,1]`
- *block* = `(acc, elem) ⇒ acc + elem`

```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Alors l'appel `reduce(block, init, list)` effectue le calcul suivant :



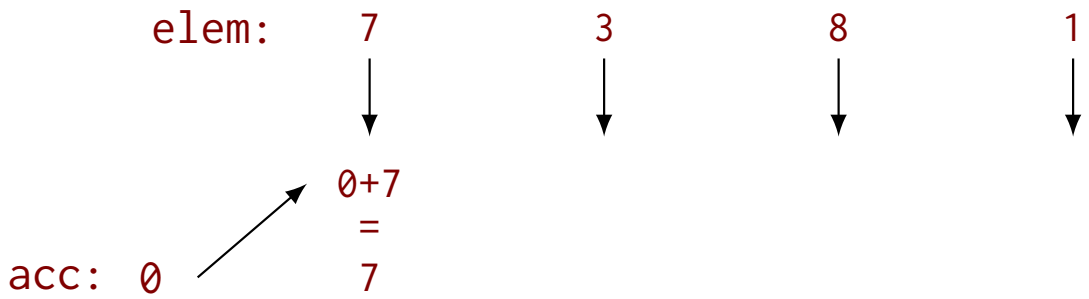
Ordre supérieur : les pliages (2/3)

Considérons l'exemple suivant d'exécution de `reduce` :

- *init* = `0`
- *list* = `[7,3,8,1]`
- *block* = `(acc, elem) ⇒ acc + elem`

```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Alors l'appel `reduce(block, init, list)` effectue le calcul suivant :



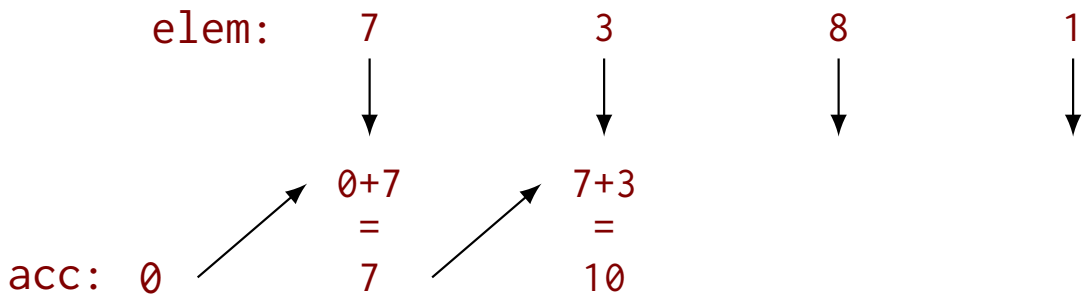
Ordre supérieur : les pliages (2/3)

Considérons l'exemple suivant d'exécution de `reduce` :

- *init* = `0`
- *list* = `[7,3,8,1]`
- *block* = `(acc, elem) ⇒ acc + elem`

```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Alors l'appel `reduce(block, init, list)` effectue le calcul suivant :



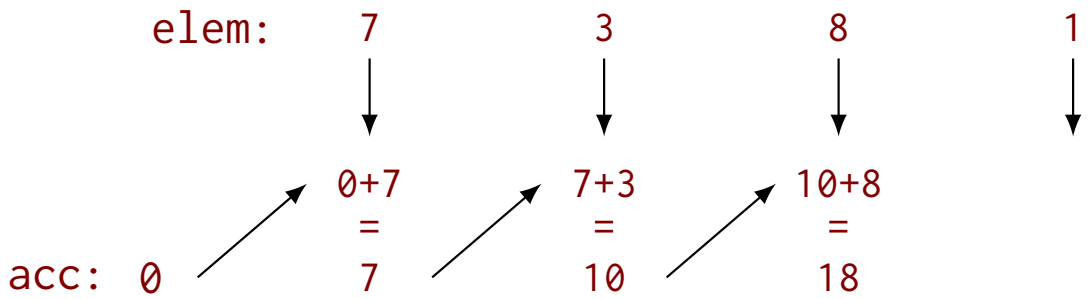
Ordre supérieur : les pliages (2/3)

Considérons l'exemple suivant d'exécution de `reduce` :

- *init* = `0`
- *list* = `[7,3,8,1]`
- *block* = `(acc, elem) ⇒ acc + elem`

```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Alors l'appel `reduce(block, init, list)` effectue le calcul suivant :



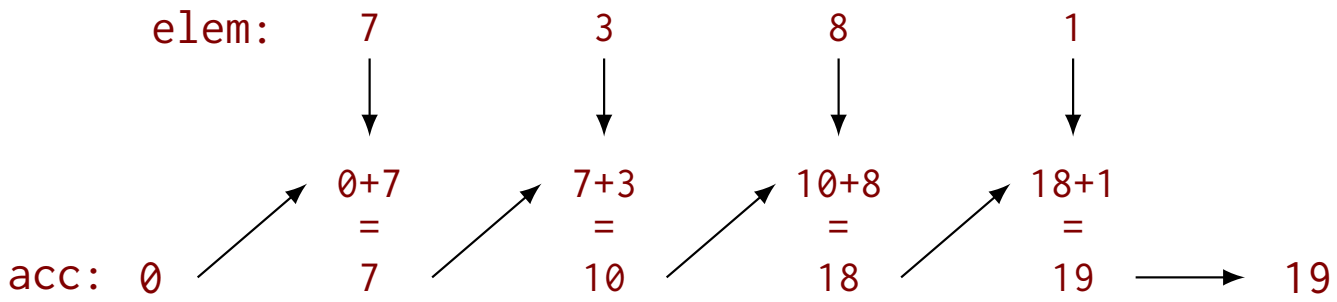
Ordre supérieur : les pliages (2/3)

Considérons l'exemple suivant d'exécution de `reduce` :

- *init* = `0`
- *list* = `[7,3,8,1]`
- *block* = `(acc, elem) ⇒ acc + elem`

```
function reduce(block, init, list) {
  if (isEmpty(list))
    return init;
  else {
    const ninit = block(init, head(list));
    return reduce(block, ninit, tail(list));
} }
```

Alors l'appel `reduce(block, init, list)` effectue le calcul suivant :



Ordre supérieur : les pliages (3/3)

Quelques exemples d'application de `reduce` (avec la bibliothèque standard)

- Sommer les éléments d'un tableau :

```
[7, 3, 8, 1].reduce((acc, el) ⇒ acc+el, 0);                    // → 19
```

- Calculer le miroir d'un tableau :

```
[7, 3, 8, 1].reduce((acc, el) ⇒ [el].concat(acc), []); // →[1,8,3,7]
```

- Compter les éléments identiques d'un tableau :

```
[1, 3, 5, 2, 3, 7, 1, 7, 3].reduce(
          (acc, el) ⇒ {
             acc[el] = (acc[el] || 0) + 1 ;
             return acc;
} , {});                                  // → { '1': 2, '2': 1, '3': 3, '5': 1, '7': 2 }
```

Ordre supérieur : les pliages (3/3)

Quelques exemples d'application de `reduce` (avec la bibliothèque `underscore`)

```
import * as _ from 'underscore';
```

- Sommer les éléments d'un tableau :

```
_.reduce([7,3,8,1], (acc, el) ⇒ acc+el, 0);          // → 19
```

- Calculer le miroir d'un tableau :

```
_.reduce([7,3,8,1], (acc, el) ⇒ [el].concat(acc), []); // →[1,8,3,7]
```

- Compter les éléments identiques d'un tableau :

```
_.reduce([1, 3, 5, 2, 3, 7, 1, 7, 3],
         (acc, el) ⇒ {
            acc[el] = (acc[el] || 0) + 1 ;
            return acc;
} , {});                          // → { '1': 2, '2': 1, '3': 3, '5': 1, '7': 2 }
```

Application : retour sur la fonction size

Reprenons le problème de calculer le nombre de sommets d'un arbre, en utilisant les opérateurs d'ordre supérieur `map` et `reduce` :

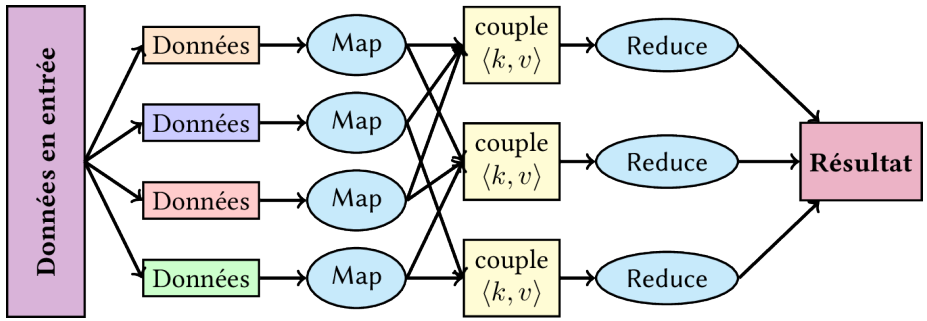
```
function treeSizeHigher(t) {
   const childrenSizes = map(treeSizeHigher, children(t));
   const sumSizes      = reduce((acc, el) ⇒ acc+el, childrenSizes, 0);
   return 1 + sumSizes;
}
```

- Les fonctions d'ordre supérieur facilitent l'écriture d'algorithmes.
- Disposer de telles fonctions sur **chaque** type de données facilite la programmation (entre autres fonctionnelle).

Application : le framework map-reduce

Idée

Profiter de l'indépendance des calculs pour obtenir du parallèlisme.



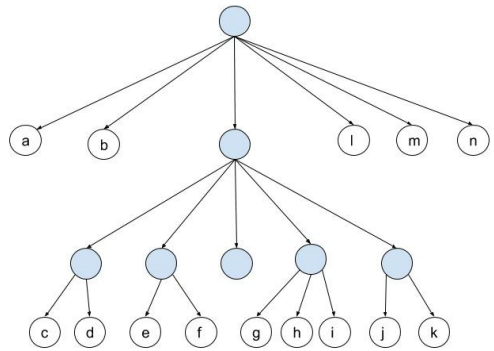
Source : Wikipedia

- Idées remontant aux années 1990 sous le nom de **skeletal programming**.
- Popularisées en particulier par Google en 2004 sous le nom MapReduce.
- Implémentées de nos jours dans des bibliothèques comme Hadoop, et facilitées dans d'autres comme OpenMP ou oneTBB.

Autres exemples de structures fonctionnelles

- Il existe pléthore de structures de données, en particulier fonctionnelles : red-black tree, trie, hash array mapped trie, finger tree, ...

Représentation du tableau
$[a, b, c, d, e, f, g, h, i, j, k, l, m, n]$
sous forme de finger tree :



Source : Wikipedia

- Ces structures tirent parti de l'immutabilité et de la persistance.
- Les complexités des opérations sur ces structures sont compétitives avec leurs équivalents impératifs. (cf. `https://www.bigocheatsheet.com`)

Conclusion

S'il y a un surcoût d'efficacité à la pureté, il peut rester raisonnable.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source : https://www.bigocheatsheet.com

Conclusion sur les types de données fonctionnels

- Les types de données inductifs permettent la programmation **pure**, en s'appuyant fortement sur de la récursivité.
- Ils peuvent être associés à des opérateurs d'ordre supérieur **génériques** capable de représenter des algorithmes complexes.
- Les propriétés de pureté permettent des optimisations remarquables : mémoire (persistance), caches (mémoïsation), parallélisation ...
- L'ensemble de ces propriétés encourage à profiter des **abstractions** : considérer d'un côté la conception optimisée des types et d'un autre côté leur utilisation par des clients.

La bibliothèque `immutable-js`

Il existe plusieurs bibliothèques `npm` orientées sur les structures immutables.
Exemple notable : la bibliothèque `immutable-js`
(`https://github.com/immutable-js/immutable-js`).

- `List`, des listes classiques
- `Map`, des dictionnaires
- `Set`, des ensembles
- `Stack`, des piles

Parmi les caractéristiques mises en avant :

- l'immutabilité, et la difficulté de vérifier les effets de bords ;
- la persistance, en réutilisant les instances existantes si possible ;
- la performance.

La bibliothèque React

React (https://react.dev) est un framework de construction d'interfaces web.
Il encourage l'utilisation de fonctions pures et d'objets immutables.

Exemple de manipulation d'un état en React

La fonction useState produit :

- une référence vers un état,
- une fonction pour modifier cet état.

Les modifications de l'état sont faites en créant un nouvel état **indépendant**.

Il est alors possible de **séparer** la gestion de l'état de celle du rendu de la page.

```
import React, { useState } from "react";

export function App() {
  const [message, setMessage] = useState("");

  return (
    <div>
      <input type="text" value={message}
        onChange={e => setMessage(e.target.value)}
      />
      <p>
        <strong>{message}</strong>
      </p>
    </div>
  );
} // Adapté depuis https://blog.logrocket.com/guide-usestate-react
```

Avantages : débogage facilité, optimisations, voyage dans le temps . . .

Quelques lectures . . .


Fogus, M.: Functional Javascript.
O'Reilly, 2013.


Narbel, P.: Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml.
Vuibert, 2005.


Reade, C.: Elements of Functional Programming.
Addison-Wesley, 1989.


Bird, R. et P. Wadler: Introduction to Functional Programming.
Prentice Hall, 1988.


Okasaki, C.: Purely functional data structures.
Cambridge University Press, 1999.


Narbel, P.: Techniques Avancées de Programmation.
Cours de Master 2 à l'Université de Bordeaux.