PG104 - Programmation Fonctionnelle

D. Renault

ENSEIRB-Matmeca

27 janvier 2025, v.1.2.5

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

Pureté, Modularité

Principe général

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

- Étudier cette question au niveau d'un ensemble de fonctions
- Proposer un critère matérialisant les dépendances ⇒ **effet de bord**
- Limiter l'impact de ces effets de bords ⇒ **pureté**
- Mettre en place des techniques pour programmer en limitant les dépendances ⇒ **récursivité**, puis **modularité**

Définition (Modularité)

Propriété logicielle selon laquelle des composants sont agencés avec peu de dépendances et facilement remplaçables par des composants équivalents.

Qu'est-ce qu'une fonction ?

Définition (Fonction – au sens mathématique)

Une fonction $D \to E$ est une correspondance univoque de D vers E.
A tout élément de D, la fonction associe un unique élément de E.

- `Math.sqrt`

```
Math.sqrt(4); // → 2
```

- `Date.parse`

```
Date.parse("2021-01-01"); // → 1609459200000
```

Définition (Fonction – au sens informatique)

Une fonction $D \to E$ est la représentation d'un calcul paramétré par un élément de D, dépendant possiblement de facteurs externes, produisant un élément de E.

- `Math.random`

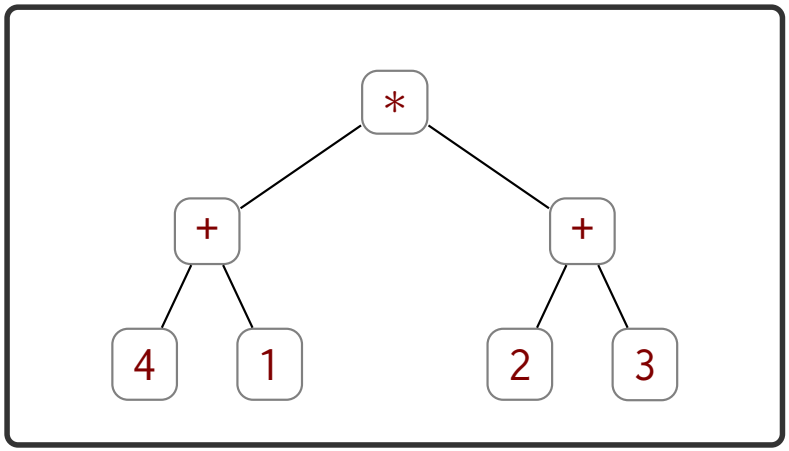
```
Math.random(); // → a number between 0 and 1
```

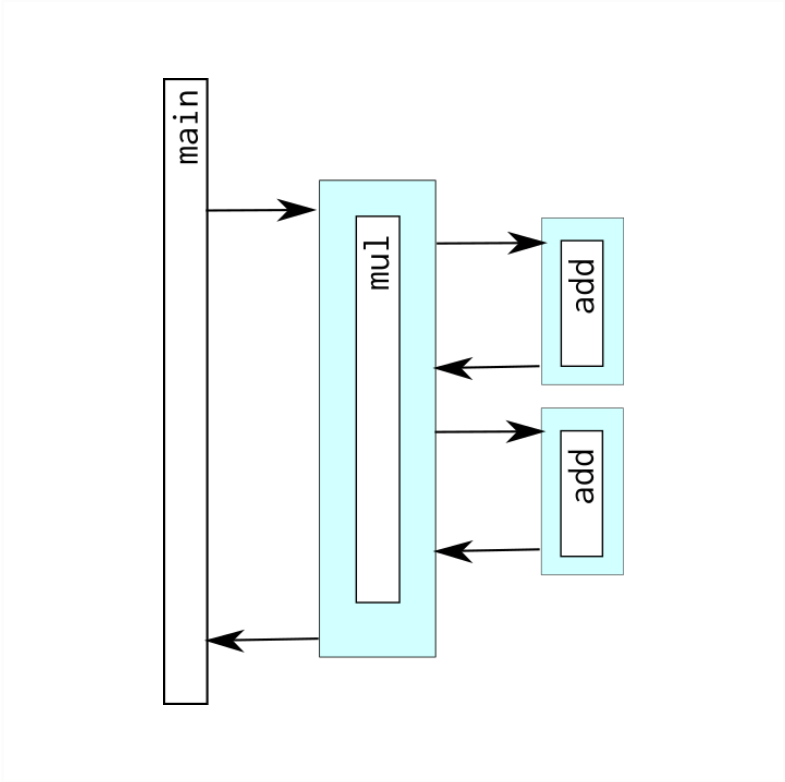
- `Date.now`

```
Date.now(); // → a positive number
```

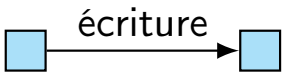
Cette différence de philosophie se traduit donc dans les programmes.

```
function main() {
  return mul(add(4, 1),
             add(2, 3));
}
```



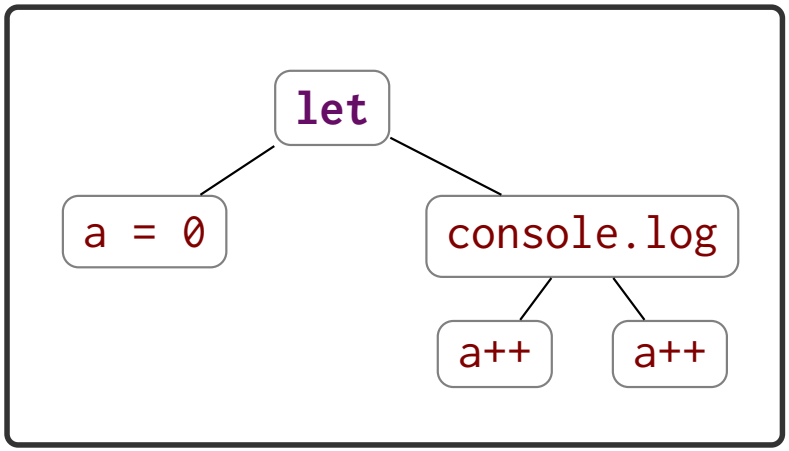


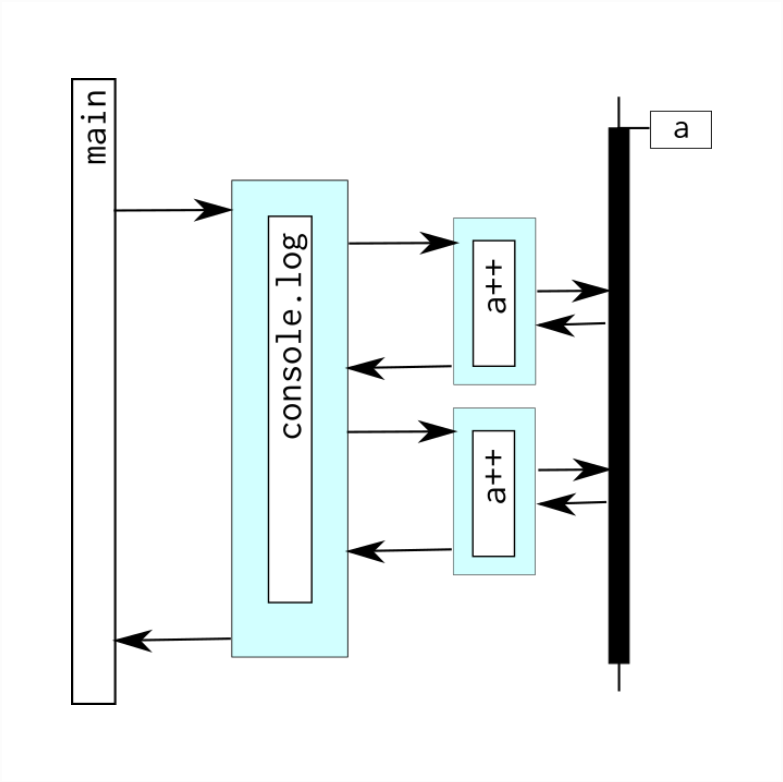
Les flèches indiquent les **dépendances** que les calculs entretiennent entre eux.



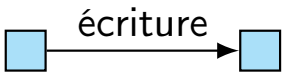
Cette différence de philosophie se traduit donc dans les programmes.

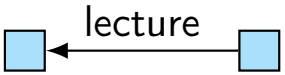
```
function main() {
  let a = 0;
  console.log(a++, a++);
}
```





Les flèches indiquent les **dépendances** que les calculs entretiennent entre eux.





Effet de bord

Définition (Effet de bord)

Un **effet de bord** (*side-effect*) est un phénomène qui se produit lorsque, lors de l'évaluation d'une expression, l'environnement dans lequel se fait l'évaluation est modifié.

Exemples d'effets de bords

- Modification d'une variable, `array.push`
- Lecture d'un variable en mouvement, `Date.now`
- Écriture dans un canal/stream. `console.log`

Les effets de bords matérialisent la différence entre une fonction au sens mathématique et au sens informatique.

Effets de bords sur les tableaux

Les manipulations de tableaux, des structures de données **mutables**, sont propices aux effets de bords, e.g. par modifications en place (*in-place*).

- Ajout d'un élément à un tableau :

```
let anArray = [1,2];
// addition producing a new independent array
let anotherArray = anArray.concat([3]);
[anArray,anotherArray]; // → [[1,2],[1,2,3]]
```

```
let anArray = [1,2];
// addition updating the array in-place
anArray.push(3);
anArray; // → [1,2,3]
```

- Retrait d'un élément d'un tableau :

```
let anArray = [1,2,3];
// removal producing a new independent array
let anotherArray = anArray.slice(1);
[anArray, anotherArray]; // → [[1,2,3],[2,3]]
```

```
let anArray = [1,2,3];
// removal updating the array in-place
anArray.splice(0, 1);
anArray; // → [2,3]
```

Problèmes des effets de bord

Principe

Les effets de bords ont tendance à compliquer la programmation.
En effet, le résultat de l'évaluation d'une expression dépend du contexte.

Exemples de complications

- Tester/prouver du code demande à gérer rigoureusement les contextes ;
- Exécuter un code dans des contextes différents ou en parallèle d'un autre code peut poser des problèmes de dépendance.

Comment programmer en limitant les problèmes dûs aux effets de bords ?

- Cloisonner les effets pour limiter les dépendances ;
- Éliminer les effets de bords au maximum $\Rightarrow$ la **pureté**.

Cloisonner les effets de bords

```
function countChars(str, c) {
    let res = 0;
    for (let i = 0; i < str.length; i++) {
        if (str[i] === c)
          res += 1;
    }
    return res;
}
```

- Cette fonction réalise des effets de bords au sein de son calcul.
- Ces effets de bords restent **internes** à la fonction `countChars`.
- Les clients de `countChars` ne sont pas impactés par ces effets de bord.

Principe de compartimentage

Cloisonner au maximum les effets de bords lorsqu'on ne peut les éviter.

Cloisonner les effets de bords

```
function countChars(str, c) {
    let res = 0;
    for (let i = 0; i < str.length; i++) { // side-effect : modification of i
        if (str[i] === c)
          res += 1;                        // side-effect : modification of res
    }                                      // end of scope of i
    return res;                            // end of scope of res
}
```

- Cette fonction réalise des effets de bords au sein de son calcul.
- Ces effets de bords restent **internes** à la fonction `countChars`.
- Les clients de `countChars` ne sont pas impactés par ces effets de bord.

Principe de compartimentage

Cloisonner au maximum les effets de bords lorsqu'on ne peut les éviter.

Pureté

Définition (Pureté)

Une fonction est dite **pure** si c'est une fonction au sens mathématique. À partir des mêmes entrées, l'évaluation d'une fonction pure doit toujours produire les mêmes résultats, **sans effet de bord**.

Exemple

- La version impérative de la fonction `countChars` n'est pas une fonction pure, elle réalise des effets de bords au sein de sa boucle.
- La version récursive de `countChars` est une fonction pure.

Définition (Transparence référentielle)

Une expression est dite **référentiellement transparente** (*referentially transparent*) si elle peut être remplacée par le résultat de son évaluation sans modifier le comportement du programme.

Exemples

- `(r) ⇒ (4*Math.pi/3) * r**3` contient une sous-expression ref. transp. et peut être remplacée par `(r) ⇒ 4.1887902047863905 * r**3`
- Les **deux** versions de `countChars` sont référentiellement transparentes.
- `Date.now() - Date.now()` ne peut pas être remplacée.

Fait

Une expression faite de fonctions pures est référentiellement transparente.

Exemples : pur ou impur ?

```
function reverseA(a) {
   if (a.length <= 1)
      return a;
   else
      return reverseA(a.slice(1)).
                concat([a[0]]);
}
```

```
function reverseB(a) {
   const b = [];
   for (let i = 0; i<a.length; i++)
      b.unshift(a[i]);
   return b;
}
```

```
function reverseC(a) {
   const b = [];
   while (a.length > 0)
      b.unshift(a.shift(0));
   return b;
}
```

```
const anArr = Array.from("abc");
reverseA(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseB(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseC(anArr); // → ['c','b','a']
anArr;           // → [ ]
```

Exemples : pur ou impur ?

```
function reverseA(a) {
   if (a.length <= 1)
      return a;
   else
      return reverseA(a.slice(1)).
                concat([a[0]]);
}
```

```
function reverseB(a) {
   const b = [];
   for (let i = 0; i<a.length; i++)
      b.unshift(a[i]);
   return b;
}
```

```
function reverseC(a) {
   const b = [];
   while (a.length > 0)
      b.unshift(a.shift(0));
   return b;
}
```

```
const anArr = Array.from("abc");
reverseA(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseB(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseC(anArr); // → ['c','b','a']
anArr;           // → [ ]
```

Pure ✓

Ref. Transp. ✓

Exemples : pur ou impur ?

```
function reverseA(a) {
   if (a.length <= 1)
      return a;
   else
      return reverseA(a.slice(1)).
                concat([a[0]]);
}
```

```
function reverseB(a) {
   const b = [];
   for (let i = 0; i<a.length; i++)
      b.unshift(a[i]);
   return b;
}
```

```
function reverseC(a) {
   const b = [];
   while (a.length > 0)
      b.unshift(a.shift(0));
   return b;
}
```

```
const anArr = Array.from("abc");
reverseA(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseB(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

```
const anArr = Array.from("abc");
reverseC(anArr); // → ['c','b','a']
anArr;           // → [ ]
```

Pure ✓
Ref. Transp. ✓

Pure ✗
Ref. Transp. ✓

Exemples : pur ou impur ?

```
function reverseA(a) {
   if (a.length <= 1)
      return a;
   else
      return reverseA(a.slice(1)).
                concat([a[0]]);
}
```

```
const anArr = Array.from("abc");
reverseA(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

Pure ✓
Ref. Transp. ✓

```
function reverseB(a) {
   const b = [];
   for (let i = 0; i<a.length; i++)
      b.unshift(a[i]);
   return b;
}
```

```
const anArr = Array.from("abc");
reverseB(anArr); // → ['c','b','a']
anArr;           // → ['a','b','c']
```

Pure ✗
Ref. Transp. ✓

```
function reverseC(a) {
   const b = [];
   while (a.length > 0)
      b.unshift(a.shift(0));
   return b;
}
```

```
const anArr = Array.from("abc");
reverseC(anArr); // → ['c','b','a']
anArr;           // → [ ]
```

Pure ✗
Ref. Transp. ✗

Programmer de manière pure (1/2)

- Si une fonction est pure, il est plus simple de raisonner dessus.
- En pratique, cela simplifie les **tests**, mais aussi les **preuves**.

Exemple : preuve en Dafny / append préserve les longueurs

Application directe du remplacement appel de fonction / évaluation

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>) // A list is either Nil or a Cons with a head and a tail
```

```
function append(xs: List, ys: List): List              // In order to append two lists xs and ys
{
  match xs                                             // Depending on the value of xs
  case Nil           ⇒ ys                              // If xs is empty, return ys
  case Cons(x, rest) ⇒ Cons(x, append(rest, ys))       // Else start with the head of xs,
                                                       //       then append the tail of xs to ys
}
```

Programmer de manière pure (1/2)

- Si une fonction est pure, il est plus simple de raisonner dessus.
- En pratique, cela simplifie les **tests**, mais aussi les **preuves**.

Exemple : preuve en Dafny / append préserve les longueurs

Application directe du remplacement appel de fonction / évaluation

```
lemma appendLengthOk(l1 : List, l2 : List)
ensures length(l1) + length(l2) == length(append(l1, l2)) {
  match (l1)

  case Nil ⇒ calc {            length(append(l1, l2));
                               ==
                               length(l2); }

  case Cons(hd1, tl1) ⇒ calc { length(append(l1, l2));
                               ==
                               length(Cons(hd1, append(tl1, l2)));
                               ==
                               1 + length(append(tl1, l2));
                               == { appendLengthOk(tl1, l2); }
                               1 + length(tl1) + length(l2);
                               ==
                               length(l1) + length(l2); } }
```

```
// ← statement of the lemma
// ← depending on l1

// • if l1 is the empty list
// ||
// || by def. of append

// • if l1 has a head and a tail
// ||
// || by def. of append
// ||
// || by def. of length
// ||
// || by induction step
// ||
// || by def. of length
```

Programmer de manière pure (2/2)

- Si une fonction est pure, il est possible d'appliquer des optimisations.
- Par exemple des techniques de **réécriture** (comme de l'inlining) ou de **mémoïsation** (comme des caches).

Exemple : mémoïsation du calcul de Fibonacci

Mise en cache des calculs déjà réalisés pour éviter de les réitérer.

```
function fibo(n) {
  if (n <= 1) { return 1n; }
  return fibo(n-1)+fibo(n-2);
}
```

$\Rightarrow$

```
{

  function fibo(n) {

    if (n <= 1) return 1n;
    return fibo(n-1)+fibo(n-2);
  }


}
```

Programmer de manière pure (2/2)

- Si une fonction est pure, il est possible d'appliquer des optimisations.
- Par exemple des techniques de **réécriture** (comme de l'inlining) ou de **mémoïsation** (comme des caches).

Exemple : mémoïsation du calcul de Fibonacci

Mise en cache des calculs déjà réalisés pour éviter de les réitérer.

```
function fibo(n) {
  if (n <= 1) { return 1n; }
  return fibo(n-1)+fibo(n-2);
}
```

$\Rightarrow$

```
{
  let memo = {}; // ← cache
  function fibo(n) {
    if (memo[n]) return memo[n];
    if (n <= 1) return 1n;
    return memo[n] = fibo(n-1)+fibo(n-2);
  }


}
```

Programmer de manière pure (2/2)

- Si une fonction est pure, il est possible d'appliquer des optimisations.
- Par exemple des techniques de **réécriture** (comme de l'inlining) ou de **mémoïsation** (comme des caches).

Exemple : mémoïsation du calcul de Fibonacci

Mise en cache des calculs déjà réalisés pour éviter de les réitérer.

```
function fibo(n) {
  if (n <= 1) { return 1n; }
  return fibo(n-1)+fibo(n-2);
}
```

$\Rightarrow$

```
function makeFibo() {
  let memo = {}; // ← cache
  function fibo(n) {
    if (memo[n]) return memo[n];
    if (n <= 1) return 1n;
    return memo[n] = fibo(n-1)+fibo(n-2);
  }
  return fibo;
}
let fibo = makeFibo();
```

Que demande un style de programmation qui ne fait aucun effet de bord ?

- pas de variables (uniquement des constantes)
- pas de boucles (uniquement des appels de fonctions)

Comment programmer une fonction non triviale sans boucles ?
Il faut faire appel à de la **récursivité**.

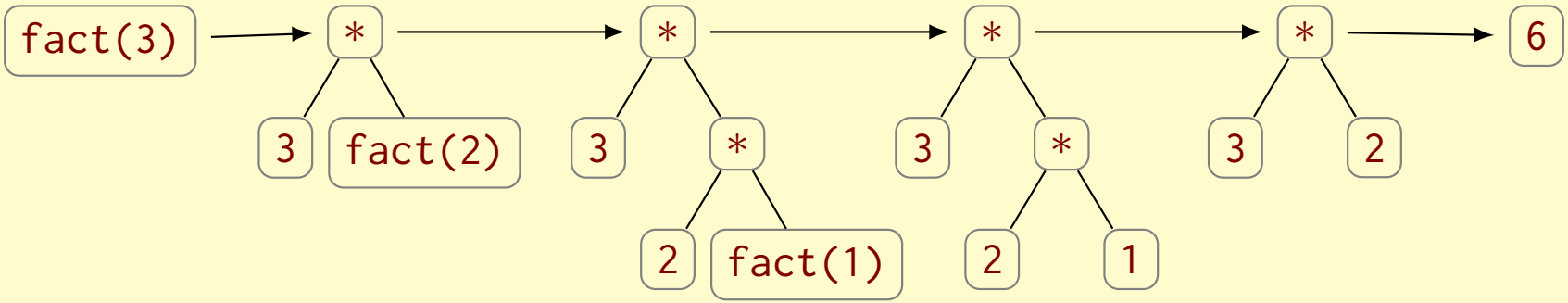
Définition (Récursif)

Un problème est dit **récursif** si il peut être décomposé en sous-problèmes *de même nature* et de taille plus petite.
Un calcul est dit **récursif** si il peut être décomposé en un nombre fini d'étapes et un nombre fini de calculs *de même nature* plus petits.

Exemple

Le calcul de la fonction factorielle $fact(n) = \prod_{k=1}^{n} k$ est un calcul récursif.

$$fact(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ n * fact(n-1) & \text{sinon} \end{cases}$$



La récursivité en programmation

De nombreux problèmes peuvent se mettre sous **forme récursive** :

- objets formels (suites récurrentes, pgcd, fractales ...),
- algorithmes gloutons, programmation dynamique,
- types de données inductifs (listes, arbres ...)

La forme récursive correspond à une décomposition formelle facilitant :

- la preuve de la **terminaison** ou de la **correction** ;
- le calcul explicite de la complexité.

Et il s'agit d'une forme permettant des **optimisations** :

- techniques de compilation (défonctionnalisation, déforestation ...)
- optimisation des appels récursifs terminaux.

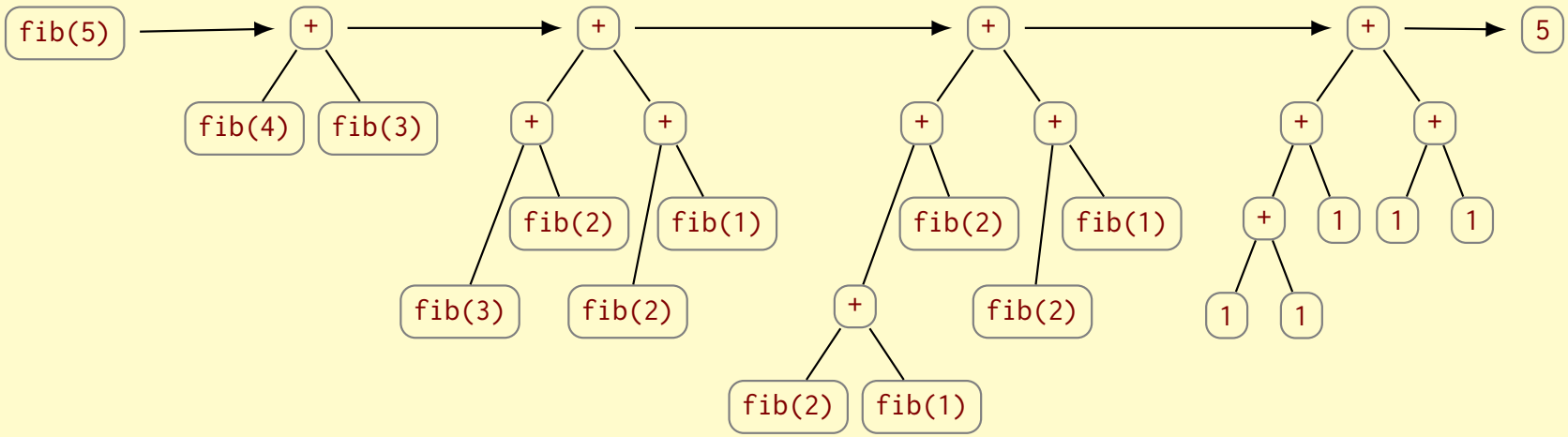
Limites des algorithmes récursifs

Les calculs peuvent devenir de taille exponentielle en leurs paramètres.
Ils peuvent ainsi dépasser la taille de la pile d'appel (*stack overflow*).

Exemple

Le calcul du n-ème terme de la suite de Fibonacci est de taille $2\,fibo(n)-1$.

$$fibo(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ fibo(n-1) + fibo(n-2) & \text{sinon} \end{cases}$$



Boucle classique vs. Fonction récursive (1/3)

Comparons une boucle classique avec un algorithme récursif “simple” :

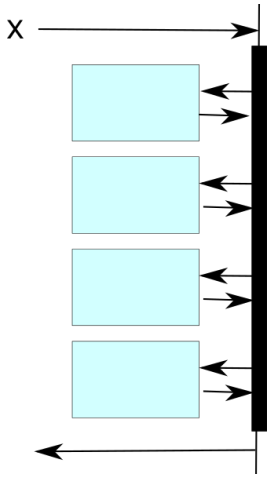
- La boucle classique :

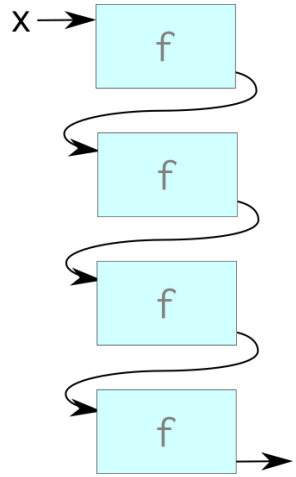
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

- L’algorithme récursif “simple” :

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

Boucle classique vs. Fonction récursive (2/3)






```
function loop(block, init, times) {
  let res = init;

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

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

```
loop((x) ⇒ x*2, 1, 10); // → 1024
```

```
recur((x) ⇒ 2*x, 1, 10); // → 1024
```

Boucle classique vs. Fonction récursive (3/3)

Comparaison des deux programmes :

- Les deux programmes sont très proches d'un point de vue fonctionnel.
- Ils n'ont pas les mêmes qualités en terme de pureté.

Quelques caveats :

- La version boucle n'est pas universelle,
  il existe d'autres formes d'algorithmes à boucles (cf. Matrix Mult) ;
- La version récursive n'est pas universelle,
  il existe d'autres formes d'algorithmes récursifs (cf. Fibonacci).

Mais au final, il n'y a pas de différence d'**expressivité** entre une boucle (simple) et une fonction récursive (simple). Juste une question de style.

Récursivité terminale

Définition (Récursivité terminale)

Un calcul est dit **récursif terminal** (*tail-recursive*) si tous ses appels récursifs sont les derniers calculs effectués par la fonction.

Exemples

Fonction récursive **non** terminale

```
function factRec(n) {
  if (n <= 1)
    return 1;
  else
    return n * factRec(n-1);
}
factRec(5); // → 120
```

Fonction récursive terminale

```
function factTr(n, r) {
  if (n <= 1)
    return r;
  else
    return factTr(n-1, n*r);
}
factTr(5, 1); // → 120
```

Récursivité terminale et réécriture

Les fonctions récursives terminales ont un fonctionnement particulier :

- soit elles renvoient une valeur finale,
- soit elles font un appel récursif qui réécrit leurs paramètres.

Elles agissent comme un **système de réécriture**.

Exemple

```
function factTr(n, r) {
  if (n <= 1)
    return r;
  else
    return factTr(n-1, n*r);
}
factTr(5, 1); // → 120
```

Réécriture des paramètres :

$$(n, r) \rightarrow (n - 1, n * r)$$

$$(5, 1) \rightarrow (4, 5) \rightarrow (3, 20) \rightarrow (2, 60) \rightarrow (1, 120) \rightarrow 120$$

Optimisation des appels récursifs terminaux

Conséquence de l'idée de réécriture

Tout algorithme récursif terminal peut être optimisé de manière à ne jamais faire exploser la pile d'appel.

Exemple : optimisation des appels récursifs dans gcc

L'option `-foptimize-sibling-calls` de `gcc` est capable de transformer :

```
int sum(int n) {
  if (n > 0)
    return n + sum(n - 1);
  else
    return 0;   }
```

en

```
int sum(int n) {
  int acc = 0;
  while (n > 0)
    acc += n--;
  return acc;   }
```

Bien que la norme Ecmascript 6 le demande (*proper tail calls*), les machines JS actuelles n'optimisent pas en général les appels récursifs terminaux.

Optimisation des appels récursifs terminaux

Conséquence de l'idée de réécriture

Tout algorithme récursif terminal peut être optimisé de manière à ne jamais faire exploser la pile d'appel.

Exemple : optimisation des appels récursifs dans gcc

L'option `-foptimize-sibling-calls` de `gcc` est capable de transformer :

```
int sum(int n, int acc) {
  if (n > 0)
    return sum(n - 1, n + acc);
  else
    return 0;   }
```

en

```
int sum(int n) {
  int acc = 0;
  while (n > 0)
    acc += n--;
  return acc;   }
```

Bien que la norme Ecmascript 6 le demande (*proper tail calls*), les machines JS actuelles n'optimisent pas en général les appels récursifs terminaux.

Fait (non-trivial)

Tout algorithme récursif peut être mis sous une forme récursive terminale.

- Les fonctions récursives terminales sont transformées en réécritures.
- Il est alors possible de leur appliquer les optimisations précédentes.
- Il est aussi possible de profiter des propriétés de la pureté.

Exemple : transformation manuelle

Partons de l'exemple de la fonction `countChars` :

```
function countChars(str, c) {
   if (str.length === 0)
      return 0;
   else if (head(str) === c)
      return 1 + countChars(tail(str), c);
   else
      return countChars(tail(str), c);
}
```

Fait (non-trivial)

Tout algorithme récursif peut être mis sous une forme récursive terminale.

- Les fonctions récursives terminales sont transformées en réécritures.
- Il est alors possible de leur appliquer les optimisations précédentes.
- Il est aussi possible de profiter des propriétés de la pureté.

Exemple : transformation manuelle

Partons de l'exemple de la fonction `countChars` :

```
function countChars(str, c) {
   if (str.length === 0)
      return 0;
   else if (head(str) === c)
      return 1 + countChars(tail(str), c);
   else
      return countChars(tail(str), c);
}
```

- Une addition est réalisée **après** l'appel récursif, le rendant non-terminal.
- Elle peut être remplacée par un paramètre additionnel représentant la somme totale.

Fait (non-trivial)

Tout algorithme récursif peut être mis sous une forme récursive terminale.

- Les fonctions récursives terminales sont transformées en réécritures.
- Il est alors possible de leur appliquer les optimisations précédentes.
- Il est aussi possible de profiter des propriétés de la pureté.

Exemple : transformation manuelle

Partons de l'exemple de la fonction `countChars` :

```
function countChars(str, c) {
   if (str.length === 0)
      return 0;
   else if (head(str) === c)
      return 1 + countChars(tail(str), c);
   else
      return countChars(tail(str), c);
}
```

```
function countCharsTr(str, c, total) {
   if (str.length === 0)
      return total;
   else if (head(str) === c)
      return countCharsTr(tail(str), c, total+1);
   else
      return countCharsTr(tail(str), c, total);
}
```

Pour un peu plus d'optimisation . . .

Des techniques de compilation automatiques existent pour tirer au maximum parti de la pureté et de style déclaratif.

Exemple : programmation fonctionnelle "sur place"

Le compilateur du langage Koka est capable de transformer un algorithme pur sur des listes en une transformation "in-place" sous conditions.

La fonction récursive non-terminale :

```
fun map(l : list<a>, f : a → b) : list<b>
  match l
    Cons(x, xs) → Cons(f(x), map(xs, f))
    Nil         → Nil
```

peut être transformée automatiquement en une fonction réutilisant la mémoire déjà allouée (si possible) :

```
void map(list_t l, function_t f, list_t* res) {
  while (is_Cons(l)) {
    if (is_unique(l)) {          // if l is not shared
      box_t y = apply(f, l→head);
      l→head = y;
      *res = l;      // update previous node in-place
      res = &l→tail;        // set the result address
      l = l→tail;    // continue with the next node
    }
    else { ... }  // slow path allocating fresh nodes
  }
  *res = Nil;
}
```

Masquage des paramètres / fermetures

- Les fonctions récursives terminales possèdent souvent des paramètres supplémentaires qui nécessitent une initialisation particulière.
- En enfermant ces fonctions, il est possible de masquer ces paramètres inutiles au client :

```
function countChars(str, c) {
  function countCharsTr(str, c, total) {              // Internal definition
     if (str.length === 0)                            //
        return total;                                 //
     else if (head(str) === c)                        //
        return countCharsTr(tail(str), c, total+1);   //
     else                                             //
        return countCharsTr(tail(str), c, total);     //
  }                                                   //
  return countCharsTr(str, c, 0);
}
```

Conclusion sur la pureté

- La **pureté** offre des propriétés logicielles intéressantes, grâce à la transparence référentielle : cache, optimisations, preuve ...
- Elle nécessite une discipline de programmation particulière : pas de variables, pas de boucles.
- Elle passe souvent par l'écriture d'algorithmes **récursifs**, là encore avec une discipline d'écriture pour profiter des propriétés.
- Un cas d'application de la pureté se trouve dans les types de données inductifs : listes, arbres ...

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.