PG104 - Programmation Fonctionnelle

D. Renault

Enseirb-Matmeca

11 mars 2025, v.1.2.5

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

Organisation du cours

- 8 séances de cours de 2h chacune
- 10 séances de TD de 2h chacune, encadrées par
  Nicolas Ducarton, Émile Naquin, Antoine Rollet et moi-même.
- Un projet de programmation s'étendant sur 3 mois
- Un examen final sur machine (*)

Page du cours et des TDs

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

(*) sauf contre-ordre

Introduction

- Qu'est-ce que la programmation fonctionnelle ?
  $\Rightarrow$ Un **style** de programmation, voire plus : un **paradigme**.
- Qu'est-ce qu'un style de programmation (ou un paradigme) ?
  $\Rightarrow$ Une façon de penser les problèmes et leurs solutions.
- Qu'est-ce que cela a à voir avec la programmation ?
  $\Rightarrow$ La façon de penser influe sur la façon de programmer.
- Pourquoi maîtriser plusieurs styles de programmmation ?
  $\Rightarrow$ Différentes **qualités** du code produit, différentes **techniques** logicielles appliquables selon les styles :

  Correction, Modularité, Abstraction, Extensibilité, Fiabilité ...

Ce que ce cours n'est pas

- Un cours de Javascript (même s'il est illustré en Ecmascript) ;
- Un cours de programmation web (même si certains exemples en sont) ;
- Un cours de programmation typée (même si cela transparaît par endroits) ;
- Un cours de programmation objet (même si il y en a tapie dans l'ombre).

Par contre, il se veut appliquable dans la plupart des langages de programmation modernes (Java, C#, C++, Python, Ruby ...).

Le choix de Javascript

- Javascript n'est pas nécessairement le premier choix de langage pour apprendre la programmation fonctionnelle. D'autres candidats existent :

  Lisp, Scheme, Haskell, OCaml, F# ...

- Néanmoins, Javascript est un langage bien adapté à la **mise en application** de la programmation fonctionnelle.
  - Il est construit en partie sur des idées tirées de Scheme, considérant les fonctions comme des valeurs centrales.
  - Il est souvent associé aux navigateurs et à leur utilisation de la programmation événementielle et asynchrone.

- Il s'agit d'un langage vivant, mature mais en pleine évolution, et doté un écosystème de bibliothèques particulièrement riche.

Les grandes lignes de ce chapitre...

- Nous allons présenter quelques exemples de problèmes algorithmiques et des façons de les résoudre (i.e les programmer).
- Nous allons discuter de la notion de **paradigme**, pour donner un sens à ce qu'est une façon de résoudre un problème.
- Nous allons appliquer cette notion à un problème algorithmique classique, et comparer plusieurs manières de le résoudre.

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

$$\Delta = b^2 - 4ac$$

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

$$\Delta = b^2 - 4ac$$
$$x_1 = (-b - \sqrt{\Delta})/(2a)$$

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

$$\Delta = b^2 - 4ac$$
$$x_1 = (-b - \sqrt{\Delta})/(2a)$$
$$x_2 = (-b + \sqrt{\Delta})/(2a)$$

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

$$\Delta = b^2 - 4ac$$
$$x_1 = (-b - \sqrt{\Delta})/(2a)$$
$$x_2 = c/(ax_1)$$

(pour les gens qui font de l'algorithmique numérique)

Un petit problème simple (1/2)

Problème (Trinôme)

Calculer les racines d'un trinôme du second degré $ax^2 + bx + c = 0$.

$$\Delta = b^2 - 4ac$$
$$x_1 = (-b - \sqrt{\Delta})/(2a)$$
$$x_2 = c/(ax_1)$$

(pour les gens qui font de l'algorithmique numérique)

Résultat : la description d'un calcul sous la forme d'un ensemble d'équations.

Un petit problème simple (2/2)

Ces équations se transforment alors naturellement en algorithme :

$$\Delta = b^2 - 4ac$$
$$x_1 = (-b - \sqrt{\Delta})/(2a)$$
$$x_2 = c/(ax_1)$$

```
function computeRoots(a, b, c) {
  let delta = b*b - 4*a*c;
  let x1 = (- b - Math.sqrt(delta))/(2*a);
  let x2 = c/(a*x1);
  return [x1, x2];
}
```

Idée générale

A partir des relations entre des objets, on peut construire des programmes.

Un problème plus complexe (1/2)

Considérons un monde constitué :

- d'une 'pièce éteinte' (pe),
- d'une 'porte fermée' (pf),
- d'un 'interrupteur' (i),
- d'un 'cerbère endormi' (ce),
- d'un 'bureau' (b),
- et d'un 'steak' (s).

Le monde en question suit les règles implicites suivantes :

$$\begin{aligned} \text{porte ouverte} &= \text{porte fermée} + \text{clé} \\ \text{pièce allumée} + \text{cerbère alerte} &= \text{pièce éteinte} + \text{interrupteur} \\ \text{cerbère endormi} &= \text{cerbère alerte} + \text{steak} \\ \text{clé} &= \text{bureau} + \text{pièce allumée} + \text{cerbère endormi} \end{aligned}$$

Problème (Escape Game)

Comment ouvrir la porte pour sortir de la pièce ?

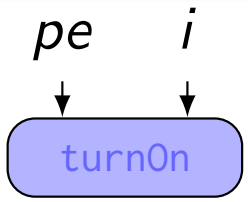
Un problème plus complexe (2/2)

$$
\begin{aligned}
ca + pa &= pe + i \\
ce &= ca + s \\
c &= b + pa + ce \\
po &= pf + c
\end{aligned}
$$

Objectif

À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)



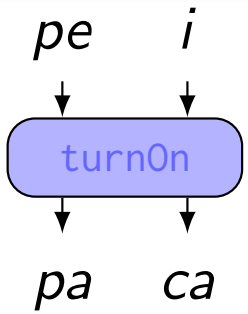
$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

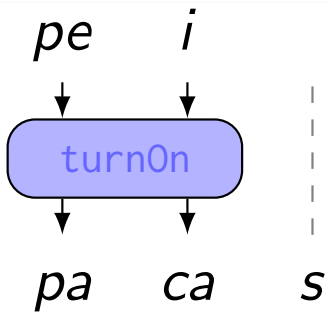


Objectif

À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

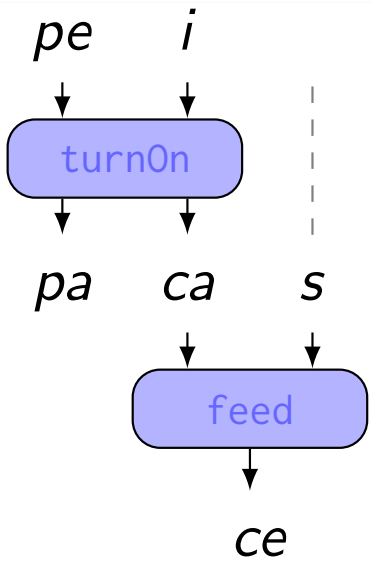


Objectif

À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

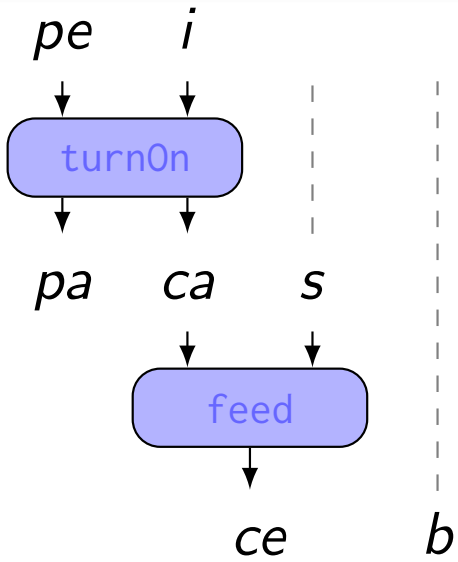


Objectif

À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$



Objectif

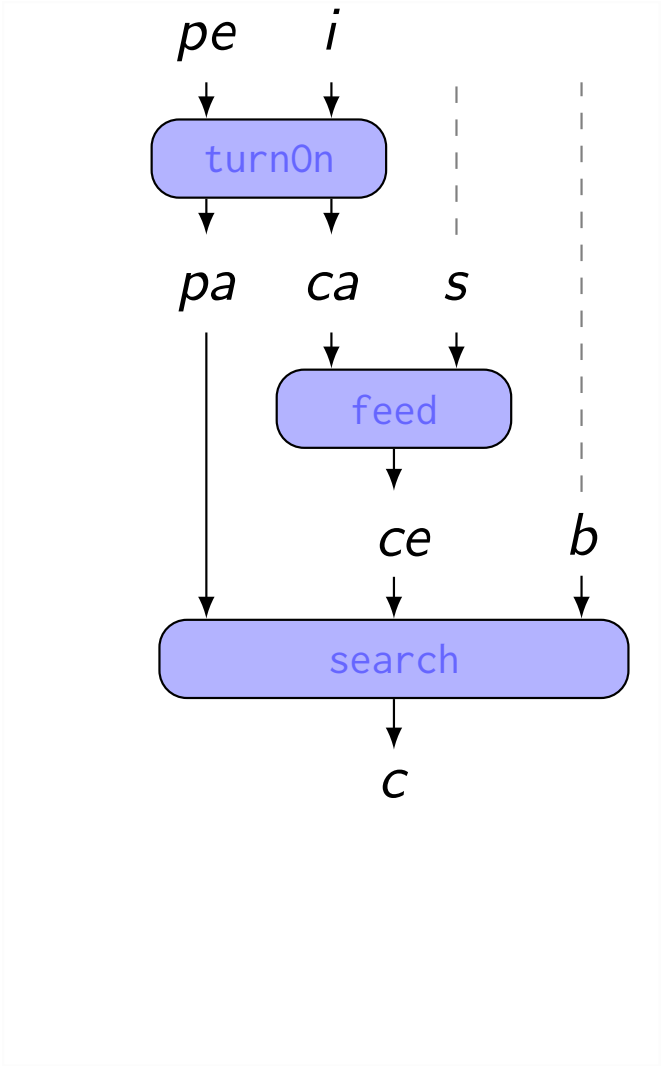
À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

À partir de pe, pf, i, b, s, produire po.

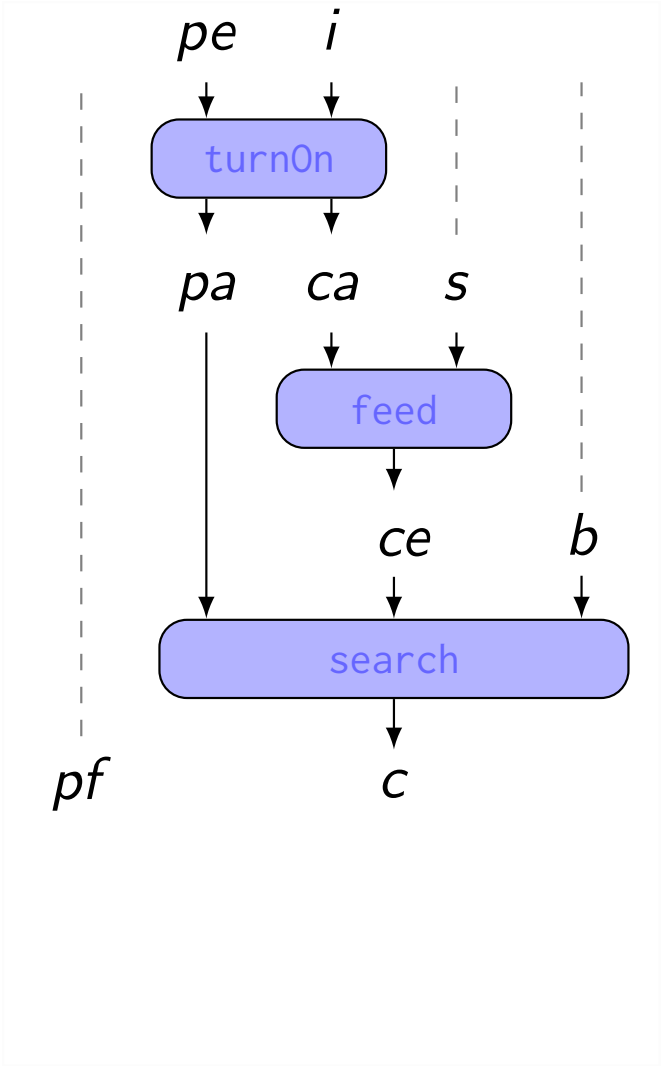


Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

À partir de pe, pf, i, b, s, produire po.

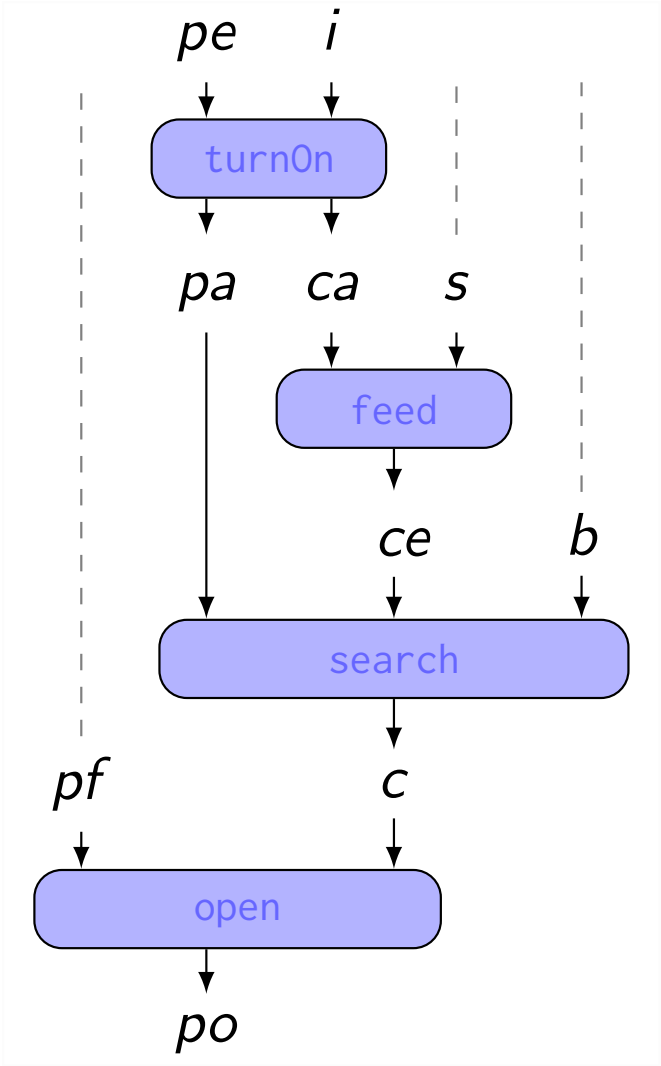


Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

À partir de pe, pf, i, b, s, produire po.



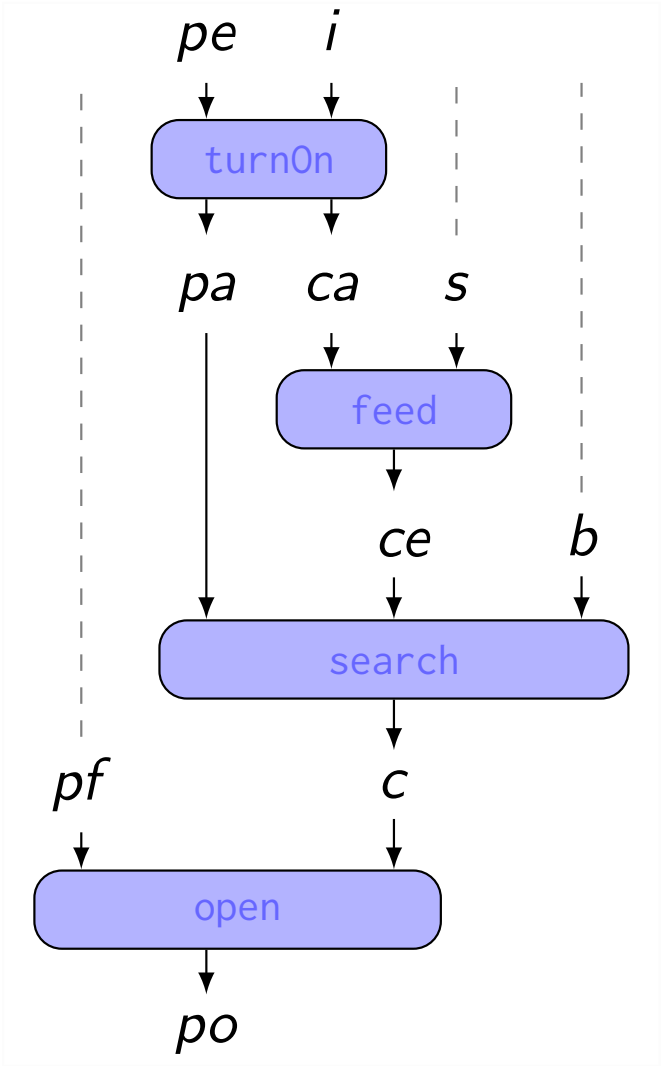
Un problème plus complexe (2/2)

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

À partir de *pe*, *pf*, *i*, *b*, *s*, produire *po*.

```
function escapeGame(pe, pf, i, b, s) {
  let [pa, ca] = turnOn(pe, i);
  let ce = feed(ca, s);
  let c  = search(pa, ce, b);
  return open(pf, c);
}
```



Examinons deux manières de modéliser une simplification du problème :

```
// Global definitions
let c : Cerberus = { isAlert: true };
let s : Steak    = { isCooked: false };

function tameCerberus() {
  if (c.isAlert && s.isCooked)
    c.isAlert = false;
}

function cookSteak() {
  s.isCooked = true;
}

// Main function
function main() {
  cookSteak();
  tameCerberus();
}
```

- Les objets sont définis de manière globale ;
- Chaque action les met à jour.

Examinons deux manières de modéliser une simplification du problème :

```
// Global definitions
let c : Cerberus = { isAlert: true };
let s : Steak    = { isCooked: false };

function tameCerberus() {
  if (c.isAlert && s.isCooked)
    c.isAlert = false;
}

function cookSteak() {
  s.isCooked = true;
}

// Main function
function main() {
  cookSteak();
  tameCerberus();
}
```

- Les objets sont définis de manière globale ;
- Chaque action les met à jour.

```
function tameCerberus(ca, s) {
  assert(ca.isAlert); assert(s.isCooked);
  let ce : Cerberus = { isAlert: false };
  return ce;
}

function cookSteak(sr) {
  let sc : Steak = { isCooked: true };
  return sc;
}

// Main function
function main() {
  let ca : Cerberus = { isAlert: true };
  let sr : Steak    = { isCooked: false };
  let sc = cookSteak(sr);
  let ce = tameCerberus(ca, sc);
}
```

- Tous les objets sont locaux et constants ;
- Chaque action les transforme en de nouveaux.

Examinons deux manières de modéliser une simplification du problème :

```
// Global definitions
let c : Cerberus = { isAlert: true };
let s : Steak    = { isCooked: false };

function tameCerberus() {
  if (c.isAlert && s.isCooked)
    c.isAlert = false;
}

function cookSteak() {
  s.isCooked = true;
}

// Main function
function main() {
  cookSteak();
  tameCerberus();
}
```

- Les objets sont définis de manière globale ;
- Chaque action les met à jour.

```
function tameCerberus(ca, s) {
  assert(ca.isAlert); assert(s.isCooked);
  let ce : Cerberus = { isAlert: false };
  return ce;
}

function cookSteak(sr) {
  let sc : Steak = { isCooked: true };
  return sc;
}

// Main function
function main() {
  let ca : Cerberus = { isAlert: true };
  let sr : Steak    = { isCooked: false };
  let sc = cookSteak(sr);
  let ce = tameCerberus(ca, sc);
}
```

- Tous les objets sont locaux et constants ;
- Chaque action les transforme en de nouveaux.

Différentes manières qui induisent différentes qualités du code final :

- Tous les objets sont uniques ;
- L'état global peut devenir incohérent.

- Tous les objets sont indépendants ;
- Bien sûr, ils ont tendance à se multiplier ;
- Mais aussi, il devient possible de les dupliquer.

Paradigmes de programmmation

Définition (Paradigme [Nar05])

Un **paradigme** est un modèle général pour décomposer les problèmes et composer des solutions. Il permet de concevoir des systèmes complexes à partir d'élements simples et composables.

- À un paradigme est usuellement associé un ensemble de **briques élémentaires** que l'on cherche à composer.
- Les paradigmes sont compris et analysés en fonctions des propriétés de leurs briques élémentaires, qui permettent à leur tour d'identifier des constructions plus complexes.

En fait, un paradigme façonne la forme des solutions apportées à un problème donné, induisant un **style** de programmation.

Des idées possibles de paradigmes

- La notion de paradigme est incontestablement abstraite, mais définit ce qu'est une méthode de résolution d'un problème de programmation.
- Wikipedia liste (de manière un peu exagérée) quelques 70 paradigmes distincts.
- En pratique, 3 paradigmes sortent du lot :
  - le paradigme **impératif**,
  - le paradigme **fonctionnel**,
  - le paradigme **objet**.
- Détaillons maintenant quelques exemples.

Programming paradigms

- Action
- Agent-oriented
- Array-oriented
- Automata-based
- Concurrent computing
  - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
  - Functional
    - Functional logic
    - Purely functional
  - Logic
    - Abductive logic
    - Answer set
    - Concurrent logic
    - Functional logic
    - Inductive logic
  - Constraint
    - Constraint logic
      - Concurrent constraint logic
  - Dataflow
    - Flow-based
    - Reactive
  - Ontology
- Differentiable
- Dynamic/scripting
- Event-driven
- Function-level (contrast: Value-level)
  - Point-free style
    - Concatenative
- Generic
- Imperative (contrast: Declarative)
  - Procedural
  - Object-oriented
    - Polymorphic

Source : Wikipedia

Le paradigme impératif

- Brique élémentaire : l'**instruction**
- Idée générale :
  - la machine possède un état (mémoire, registres, ...),
  - chaque instruction modifie cet état,
  - les instructions se composent en séquence.
- Exemple simpliste :

```
function swap(a, b) {
  let c = a;
  a = b;
  b = c;
}
```

- Représentants historiques : **Fortran** (1954), **Algol** (1958)
- Exemple emblématique : **C** (1972, dernière norme : C23 de 2024)

Le paradigme fonctionnel

- Brique élémentaire : l'**expression**
- Idée générale :
  - partir d'un ensemble de valeurs de base (numbers, strings ...)
  - les composer en expressions plus complexes à travers des opérateurs ou des fonctions.
- Exemple "simpliste" :

```
function escapeGame(pe, pf, i, b, s) {
  let [pa, ca] = turnOn(pe, i);
  let ce = feed(ca, s);
  let c  = search(pa, ce, b);
  return open(pf, c);
}
```

- Représentants historiques : **Lisp** (1958), **Scheme** (1975)
- Exemple emblématique : **Haskell** (1990, dernière norme : 2010)

Expression

Considérons un langage de programmation contenant :

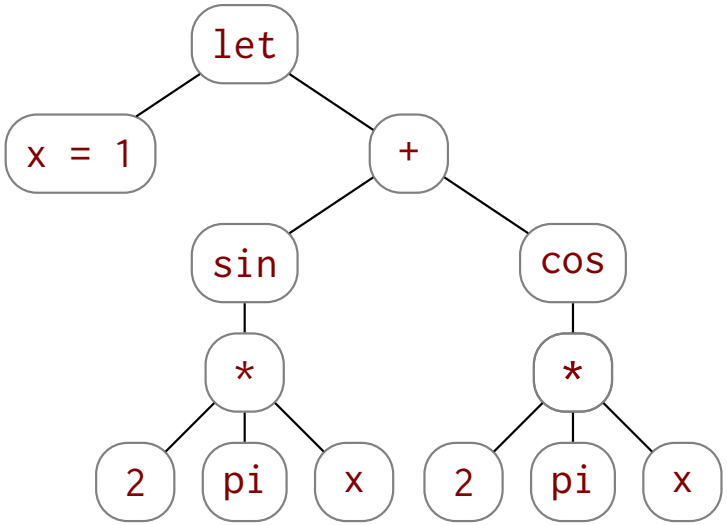
- un ensemble de **valeurs** $\mathcal{V}$ (entiers, chaînes de caractères, booléens ...) ;
- un ensemble $\mathcal{T}$ de **moyens de composer** ces valeurs (fonctions, méthodes, constructions syntaxiques particulières ...).

Définition (Expression)

Une **expression** est un arbre dans les noeuds internes sont étiquetés par des éléments de $\mathcal{T}$, et les feuilles par des éléments de $\mathcal{V}$.

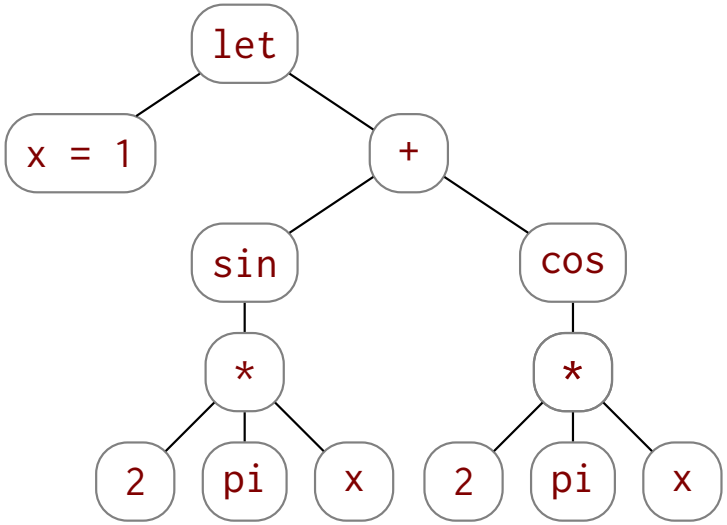
- L'expression est la représentation arborescente d'un calcul.
- **Évaluer** une expression permet d'obtenir le résultat du calcul.

Exemples d'expressions (1/2)




```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Exemples d'expressions (1/2)

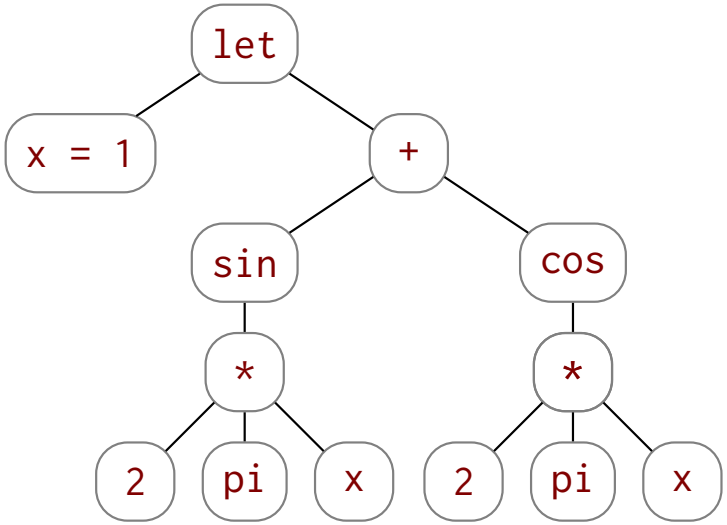



```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Haskell) :

```
let x = 1 in
  sin(2*pi*x) + cos(2*pi*x)

-- → 0.9999999999999998
```

Exemples d'expressions (1/2)

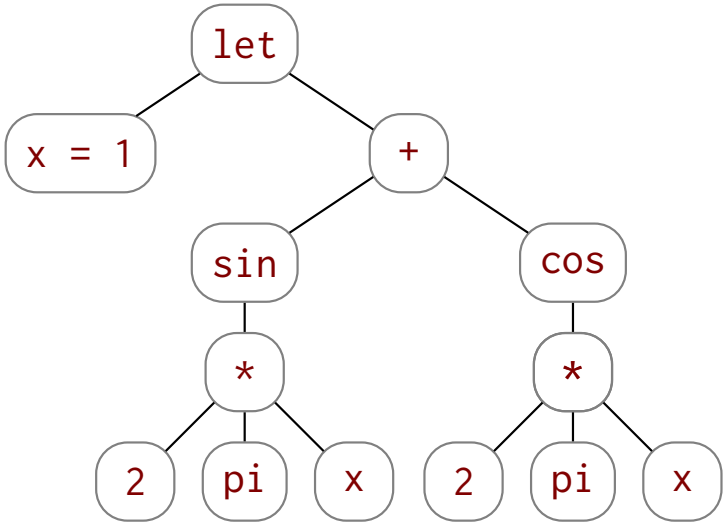



```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Javascript) :

```
let x = 1;          // should use const
Math.sin(2*Math.PI*x) +
          Math.cos(2*Math.PI*x);
// → 0.9999999999999998
```

Exemples d'expressions (1/2)



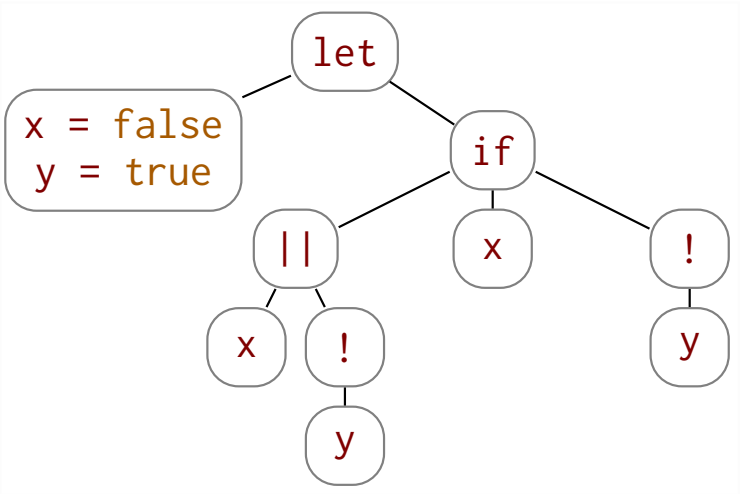

```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Javascript) :

```
let x = 1;          // should use const
Math.sin(2*Math.PI*x) +
          Math.cos(2*Math.PI*x);
// → 0.9999999999999998
```

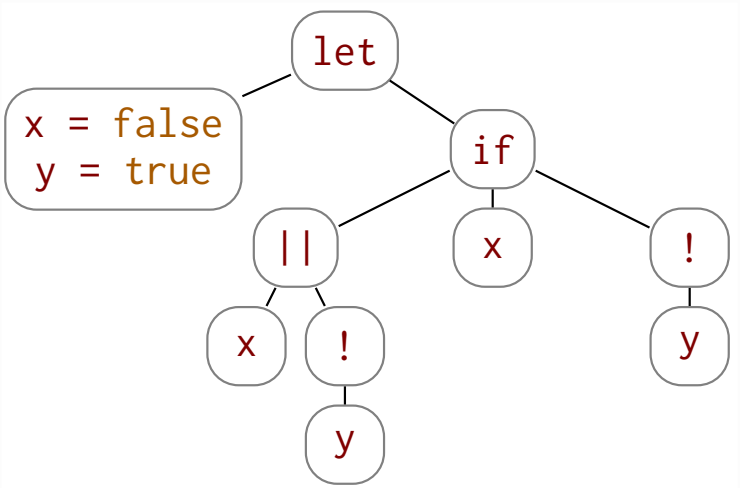
L'évaluation d'une expression se fait dans un **environnement** contenant les valeurs des objets accessibles à un moment de l'évaluation.

Exemples d'expressions (2/2)




```
let x = false and y = true in
    if (x || !y) then x else !y
```

Exemples d'expressions (2/2)



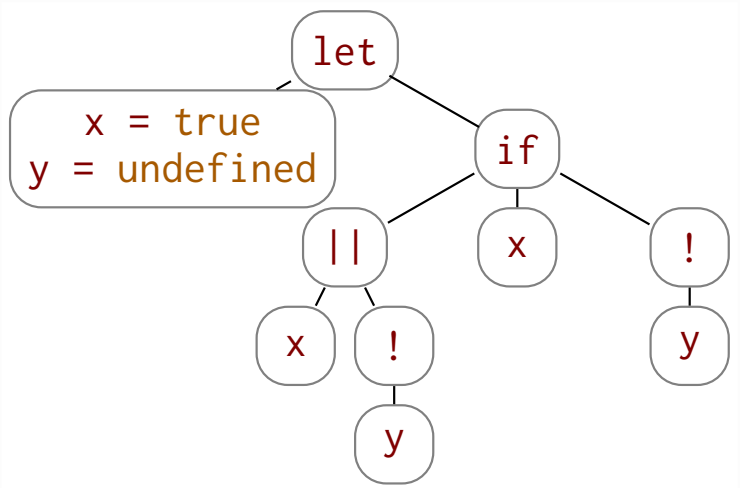

```
let x = false and y = true in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

```
x = false;
y = true;
(x || !y) ? x : !y;
// → false
```

Ternary conditional operator

Exemples d'expressions (2/2)

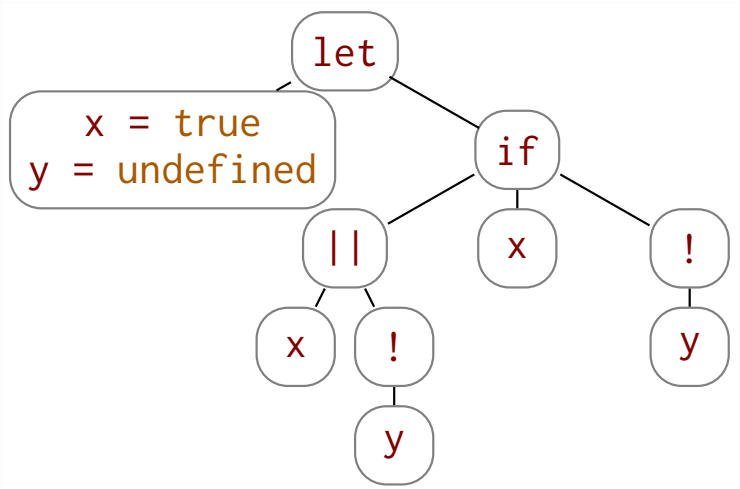



```
let x = true and y = undefined in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

```
x = true;
y = undefined;
(x || !y) ? x : !y;
// → true
```

Exemples d'expressions (2/2)




```
let x = true and y = undefined in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

```
x = true;
y = undefined;
(x || !y) ? x : !y;
// → true
```

Le résultat de l'évaluation d'une expression s'obtient en évaluant (ou pas) chacune de ses sous-expressions et composant les résultats obtenus.

Programmer uniquement avec des expressions ?

- Il existe des langages de programmation construit essentiellement sur des expressions (e.g : Lisp, Scheme, Haskell . . .)
- Mais la plupart des langages (dont Ecmascript) mélangent les expressions et les instructions.

⇒ Un style de programmation fonctionnel, de nos jours, c'est un style qui favorise la **composition** des **fonctions** pour effectuer les calculs.

- Examinons les qualités qu'on peut tirer d'un tel style de programmation, sans jamais oublier qu'il est usuellement mixé avec d'autres styles.

Exemple d'application : le comptage de caractères

Comparons les paradigmes impératif et fonctionnel sur un exemple.

Problème (Comptage de caractères)

Étant donnée une chaîne de caractères `str`, ainsi qu'un caractère `c`, compter le nombre d'occurrences de `c` dans `str`.

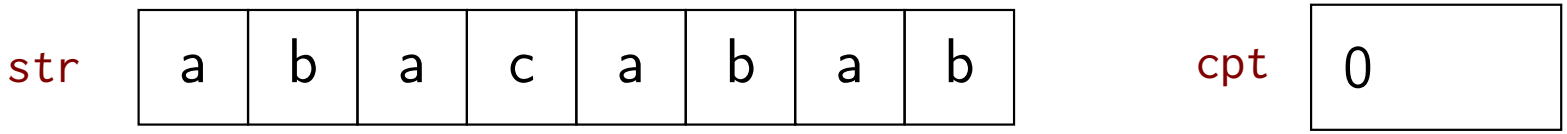
Exemples

```
countChars("",       'a');    // → 0
countChars("abacab", 'a');    // → 3
countChars("abacab", 'z');    // → 0
```

Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

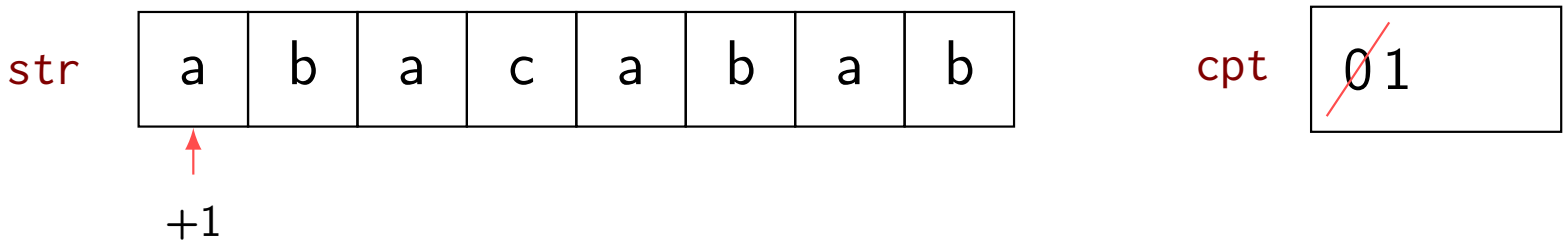
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

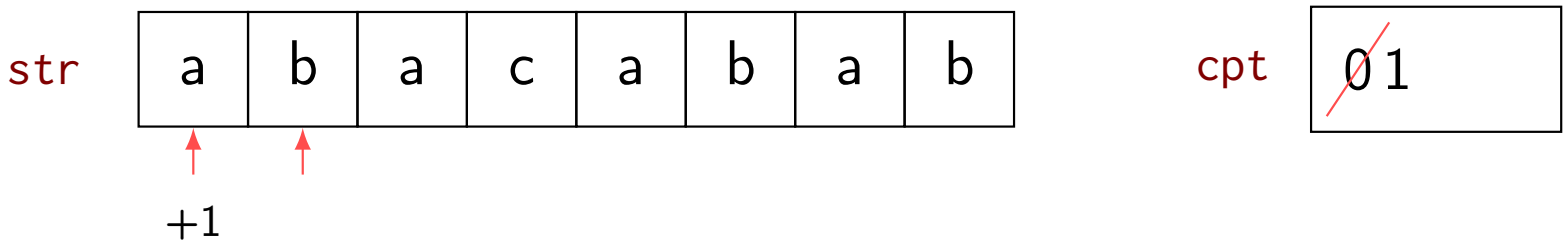
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

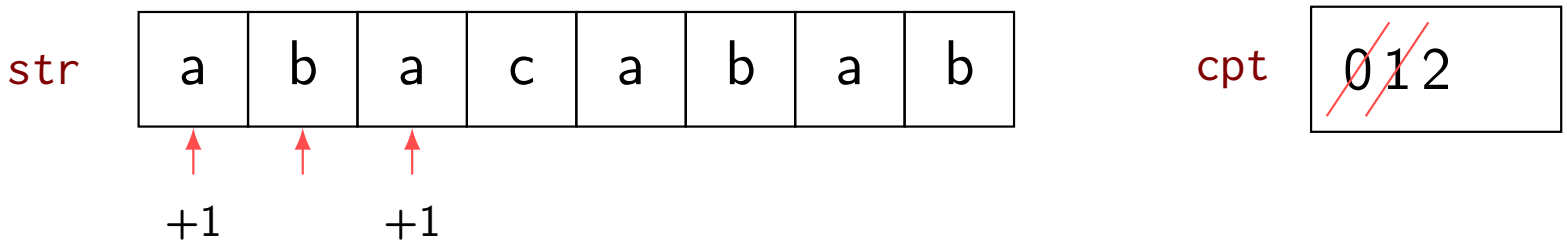
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

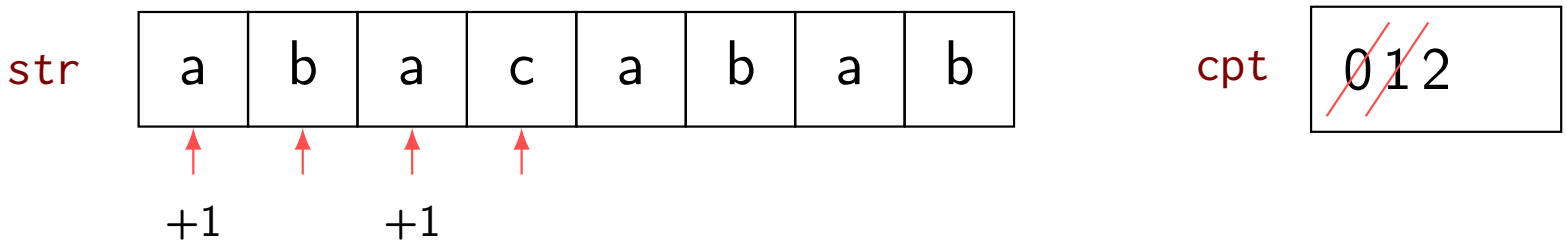
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

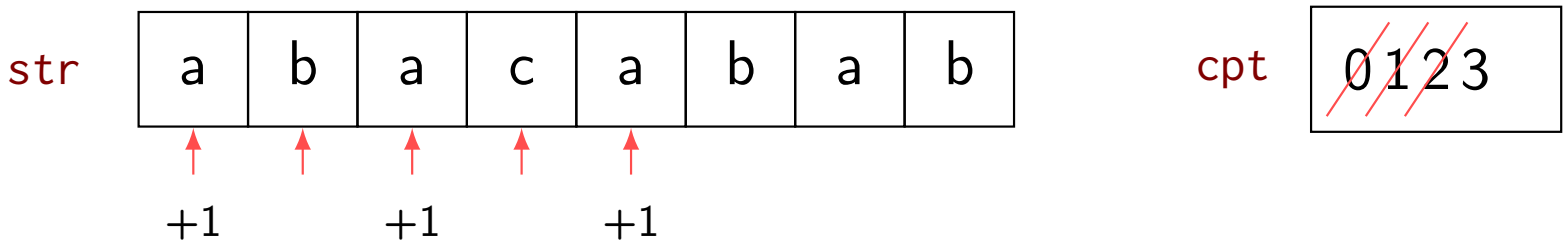
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

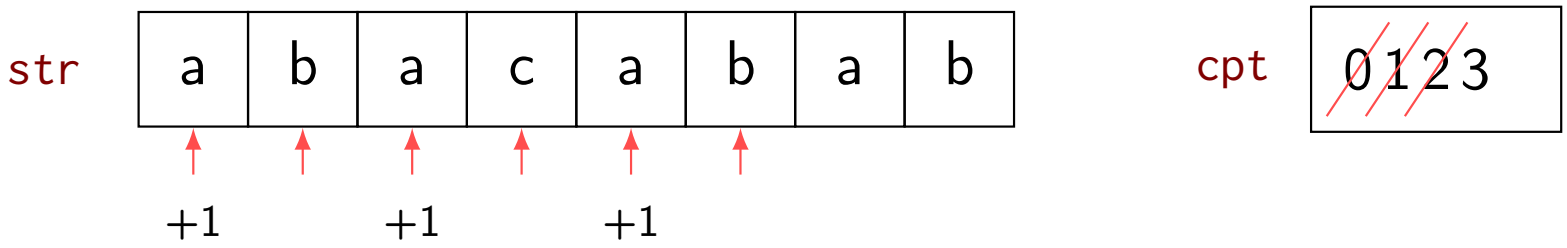
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

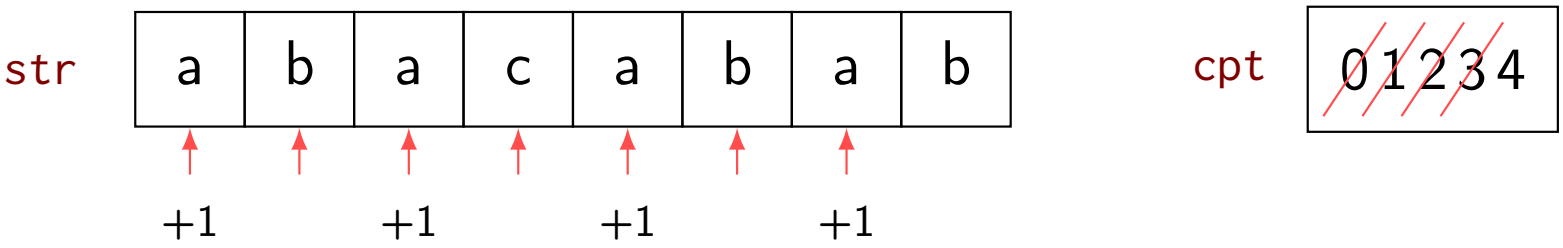
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

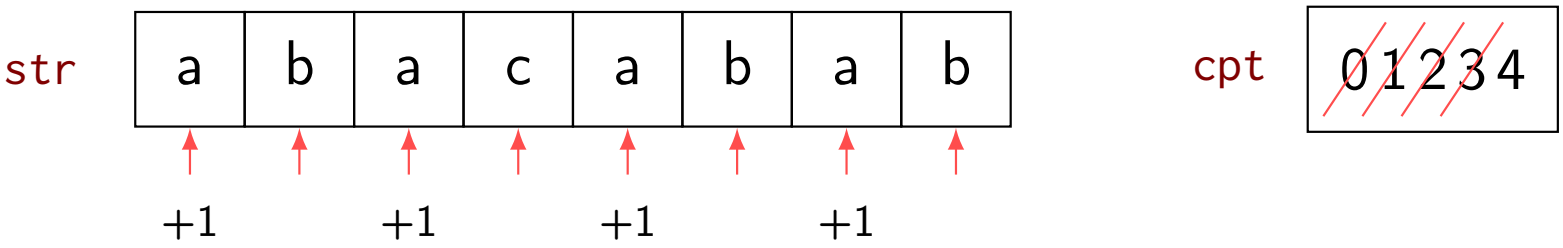
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

```
unsigned int countChars(const char* str, char c) {
  unsigned int cpt = 0;
  for (const char* it = str; *it != 0; it++)        // The const is correct
    if (*it == c)
      cpt++;
  return cpt;
}
```

- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



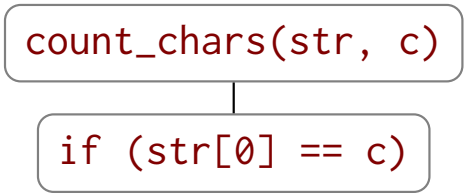
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :

```
count_chars(str, c)
```

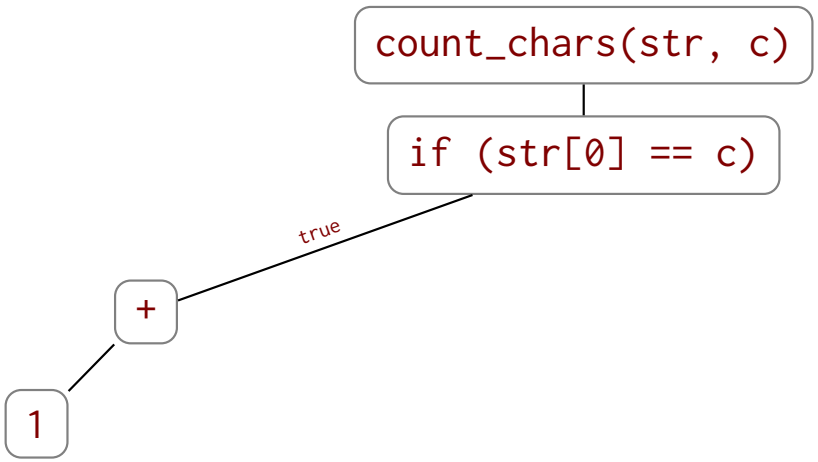
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



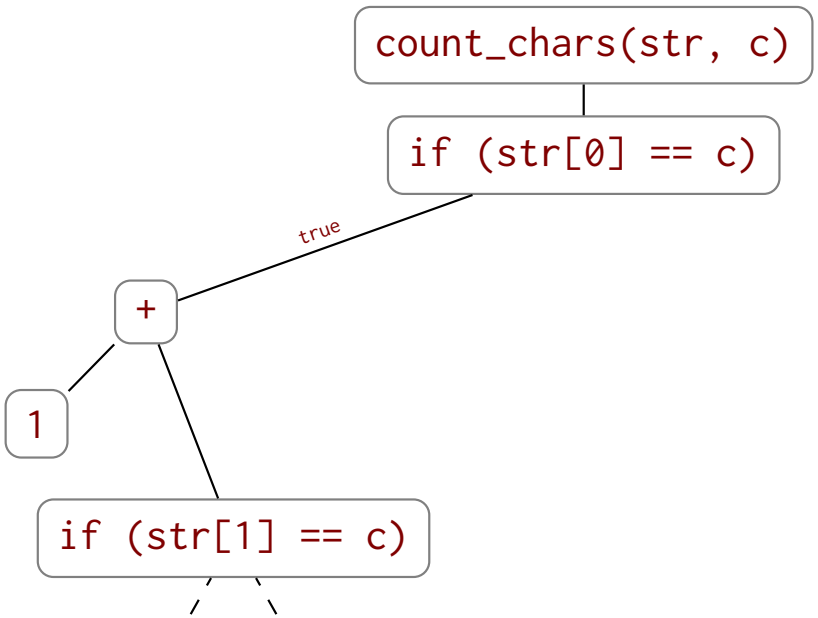
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



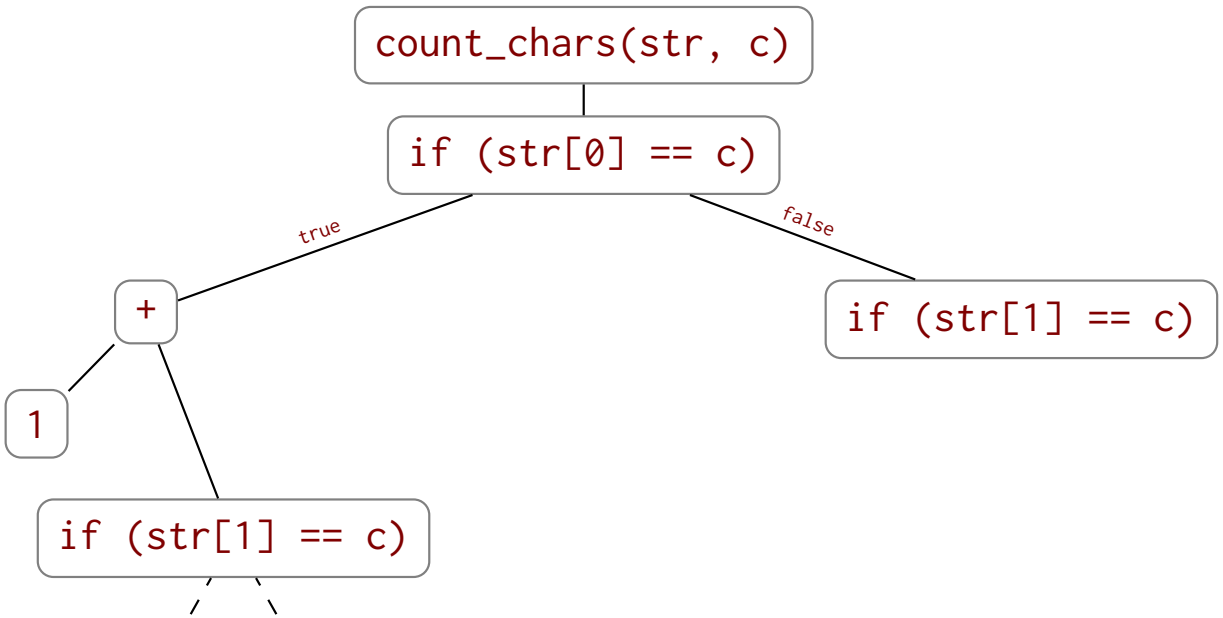
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



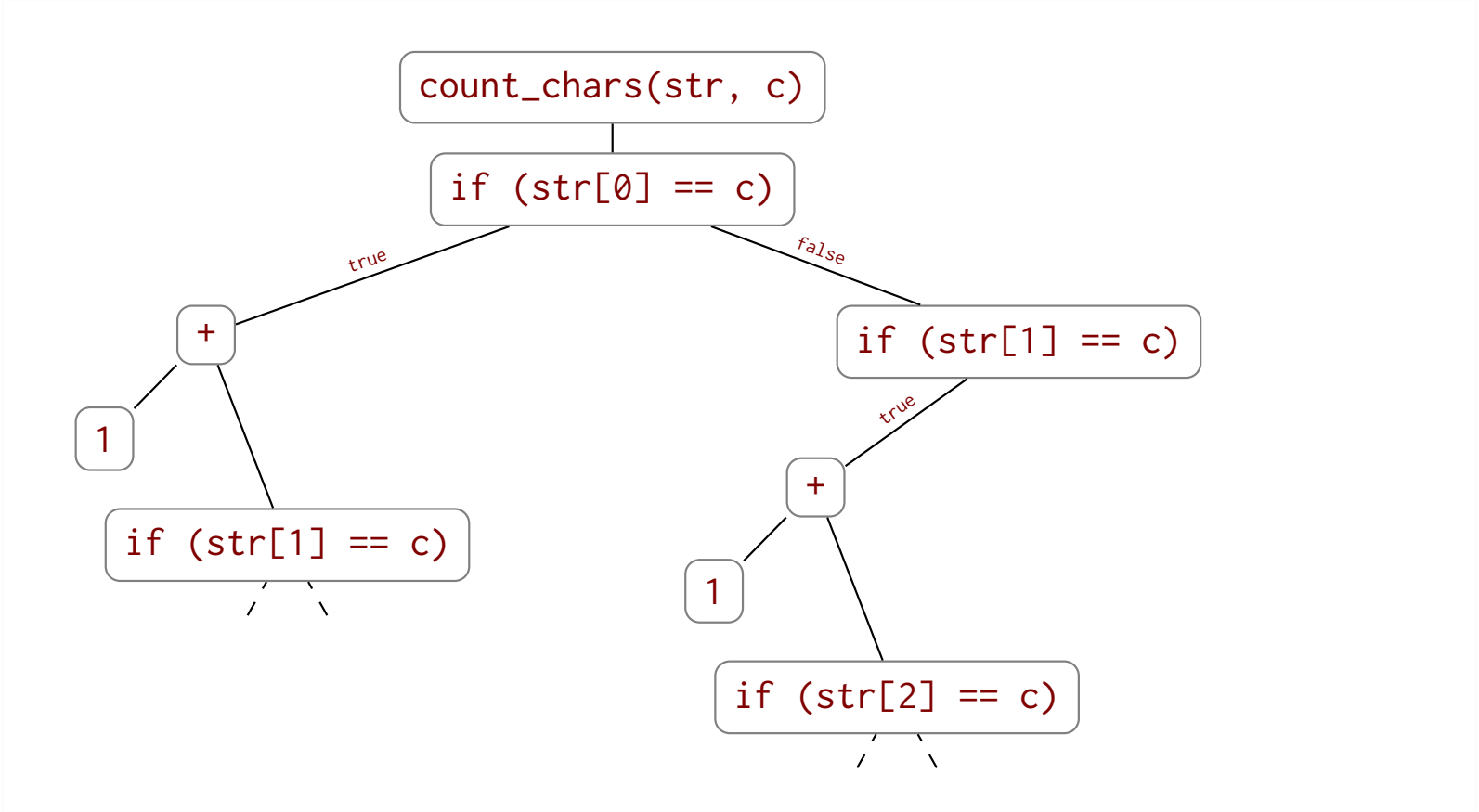
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



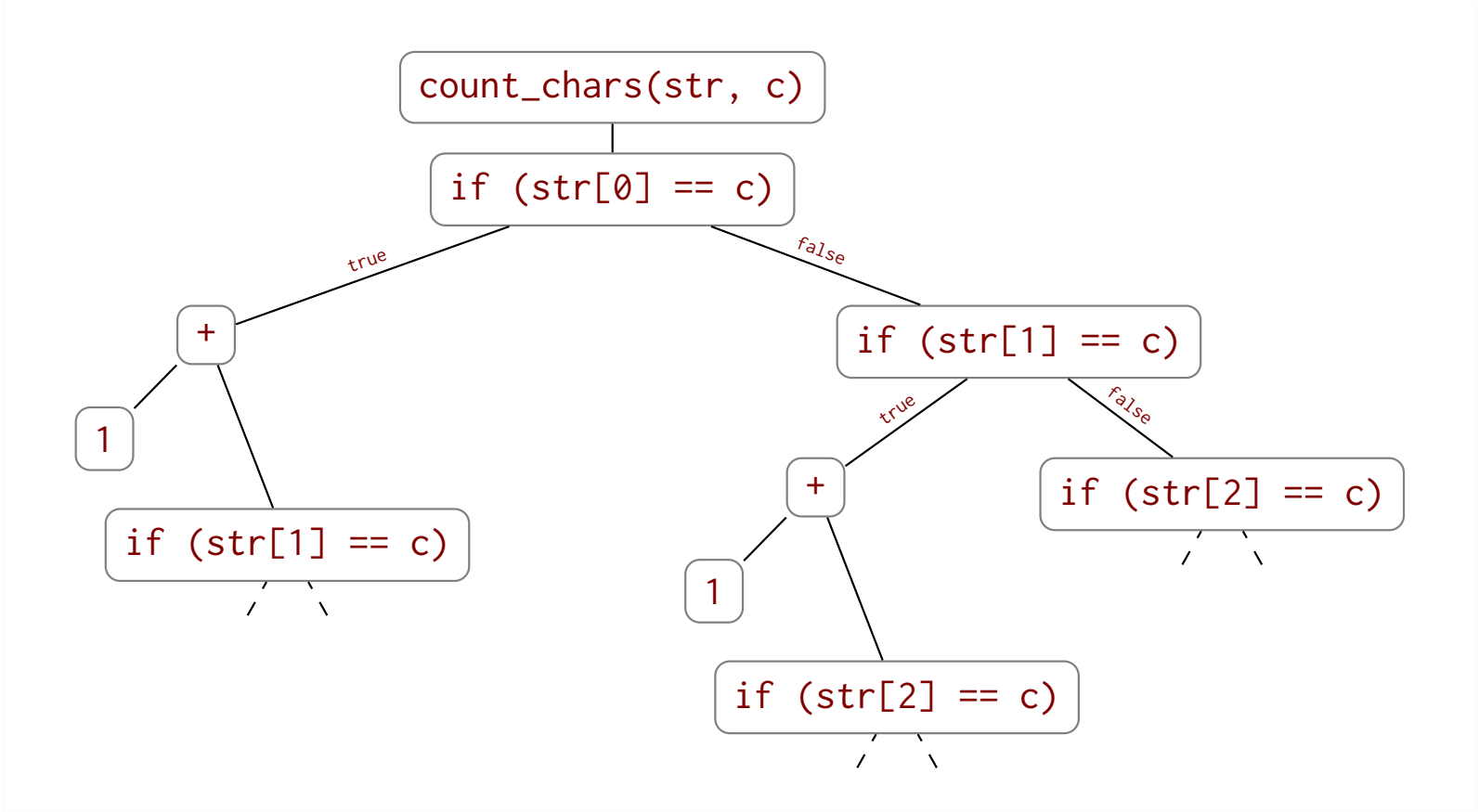
Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



Implémentation sous la forme d'une expression (1)

Pour une chaîne donnée, il est possible de créer une expression pas à pas :



Quelques remarques :

- ceci n'est pas un programme, c'est une expression qui dépend de la taille de l'entrée ;
- néanmoins, cette construction fait apparaître une décomposition naturelle du problème ;
- de plus, des parties de l'expressions identiques sont facilement reconnaissables, à quelques variations près.

Ces éléments nous invitent à résoudre le problème :

- en gérant d'abord le 1er caractère ;
- puis en écrivant une expression similaire pour une chaîne plus petite.

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Alors on peut considérer l'algorithme de la manière suivante :

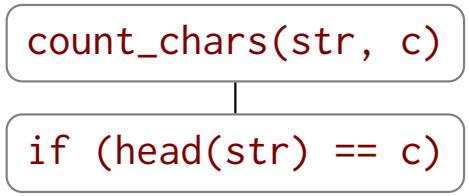
```
count_chars(str, c)
```

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Alors on peut considérer l'algorithme de la manière suivante :

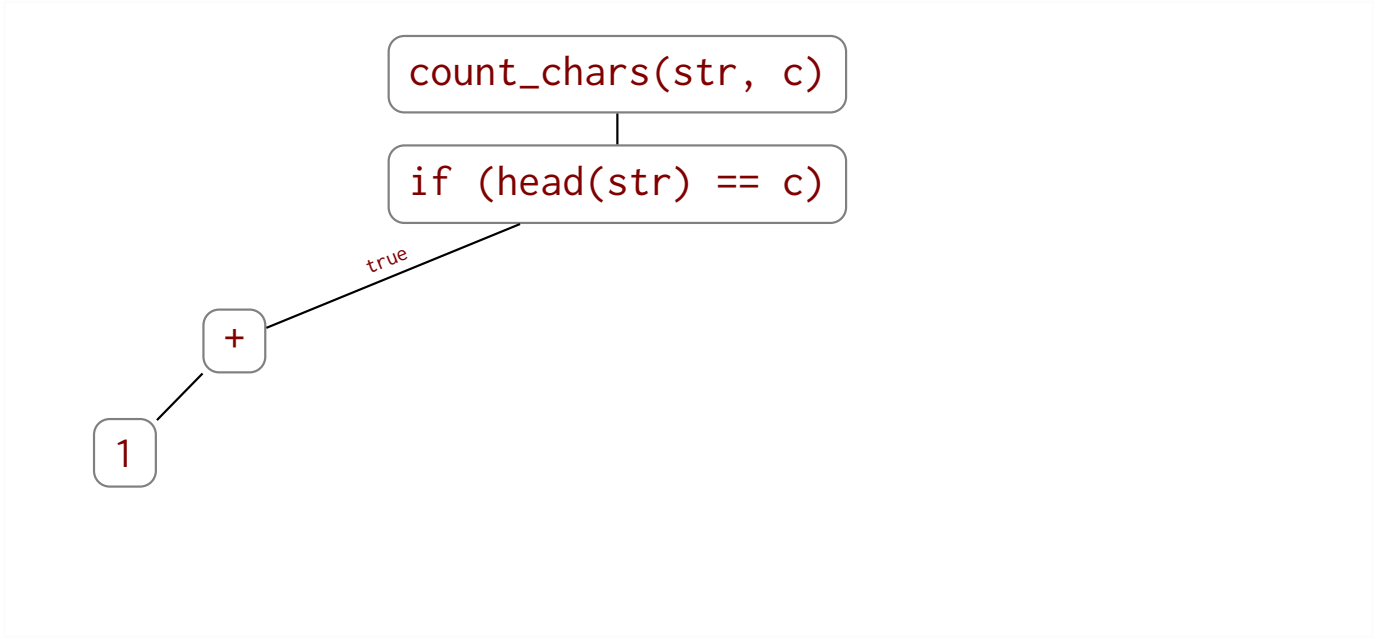


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Alors on peut considérer l'algorithme de la manière suivante :

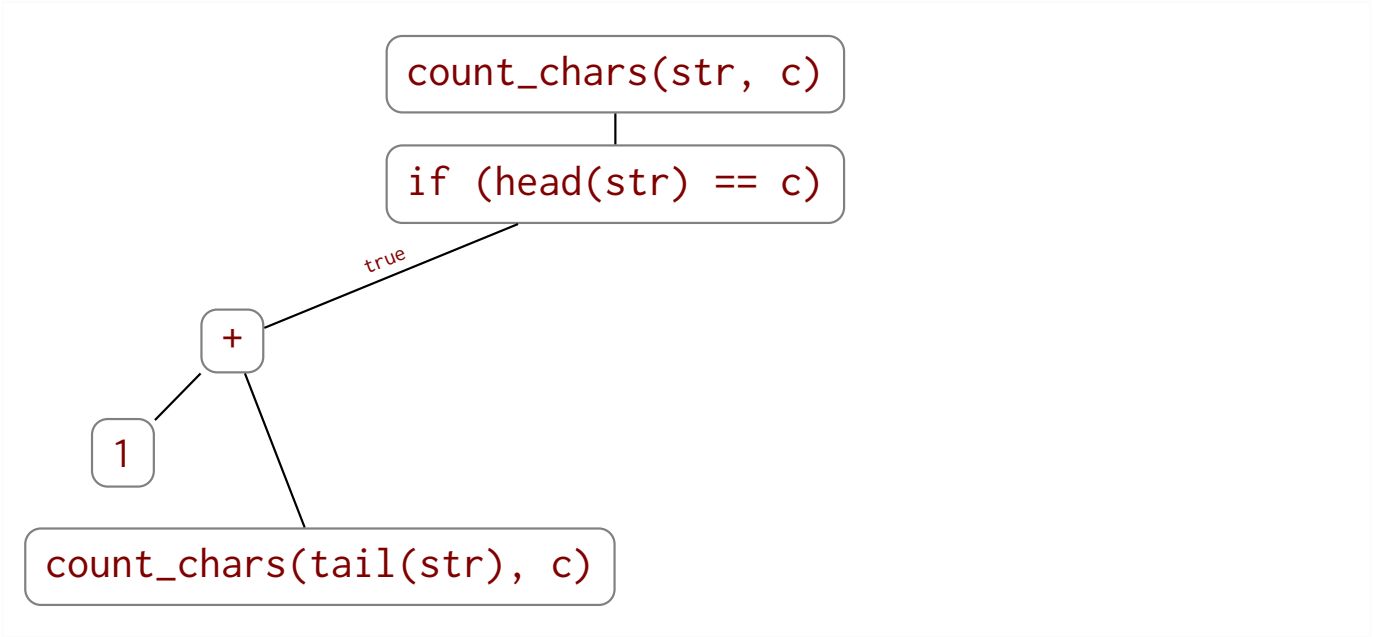


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Alors on peut considérer l'algorithme de la manière suivante :

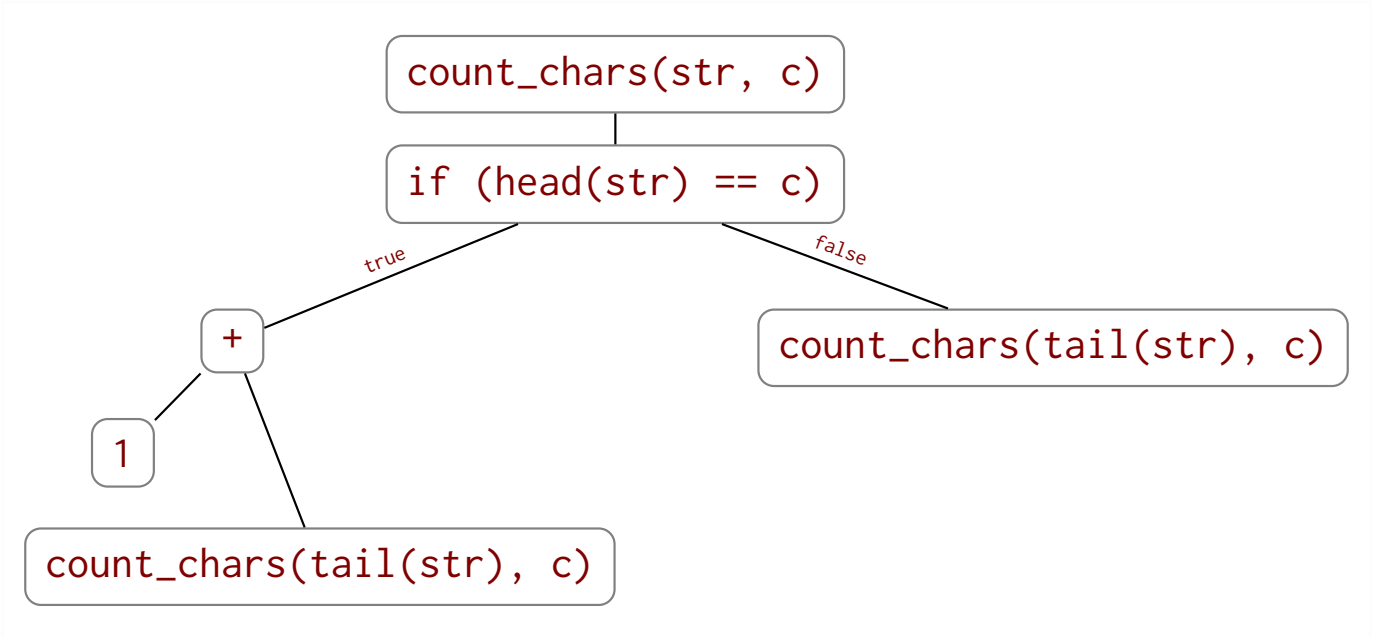


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

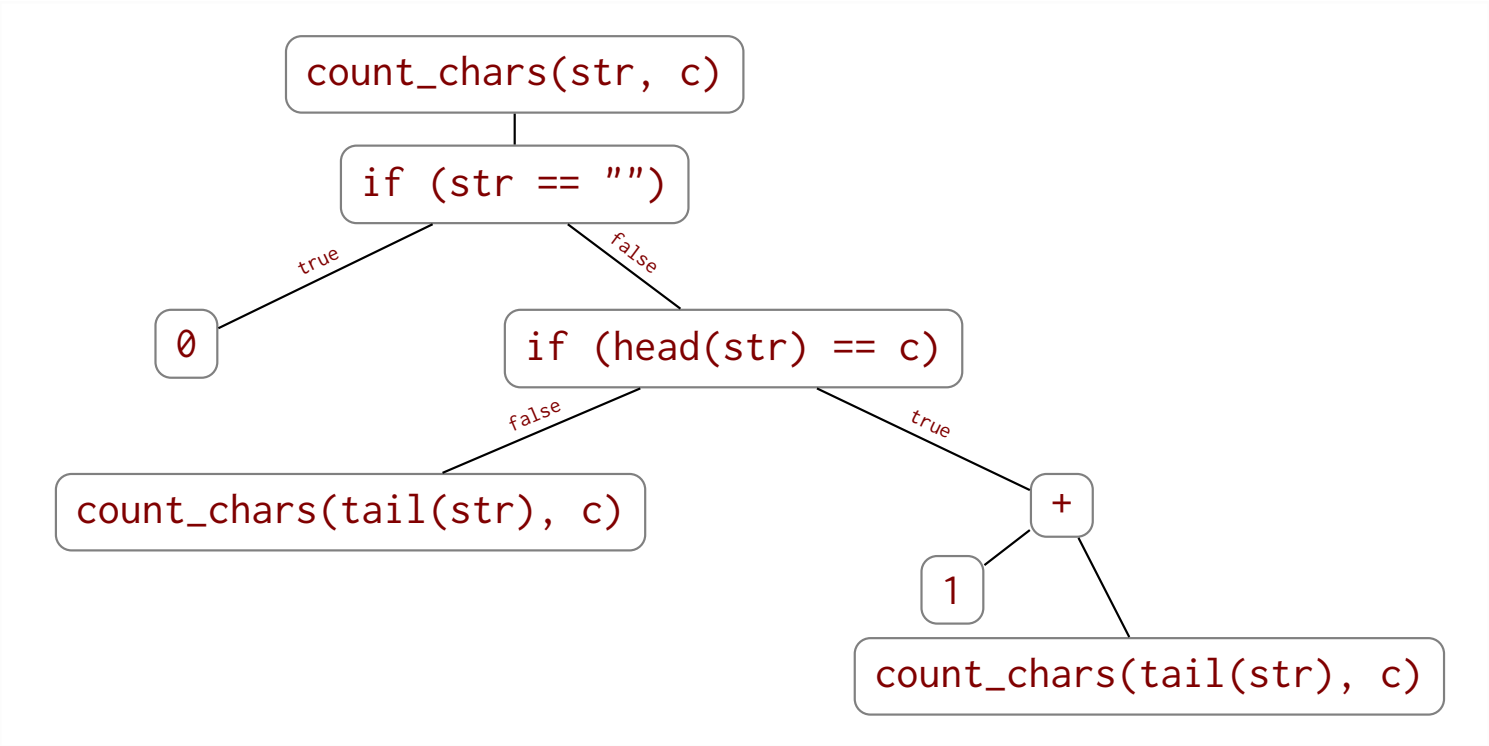
- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

Alors on peut considérer l'algorithme de la manière suivante :



L'algorithme est encore incomplet, il manque un moyen de s'arrêter.

- Il suffit pour cela d'ajouter la gestion des chaînes vides :



- La fonction construite au final est une fonction **récursive**.

Version finale Javascript

```
// Given a string ‘str‘ and a character ‘c‘, returns the number of
// occurrences of ‘c‘ inside ‘str‘.
function countChars(str, c) {
  if (str.length === 0)                    // Stopping condition
    return 0;
  else if (head(str) === c)
    return 1 + countChars(tail(str), c);
  else
    return countChars(tail(str), c);
}
```

- L'algorithme peut être décrit sous forme d'équations, ici en Haskell :

```
countChars("",  c) = 0                                  -- empty string
countChars(str, c) = if (head(str) == c)               -- any non-empty string
                     then 1 + countChars(tail(str), c)
                     else countChars(tail(str), c)
```

- Écrire des programmes comme des **ensembles d'équations**, privilégie les relations logiques entre objets plutôt que les opérations concrètes.

  On parle alors d'un style **déclaratif**.

- Penser les algorithmes en construisant des expressions, cela engendre naturellement des fonctions **récursives**.

Comparaison des styles

Version impérative (I) :

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

Version fonctionnelle (F) :

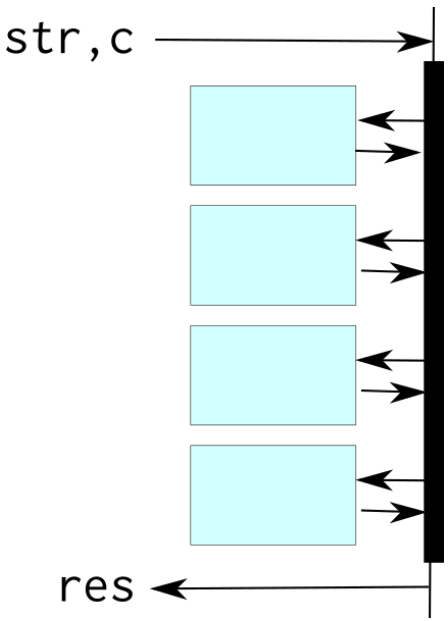
```
function countChars(str, c) {
  if (str.length === 0)
    return 0;
  else if (head(str) === c)
    return 1 + countChars(tail(str), c);
  else
    return countChars(tail(str), c);
}
```

Comparaison entre les versions :

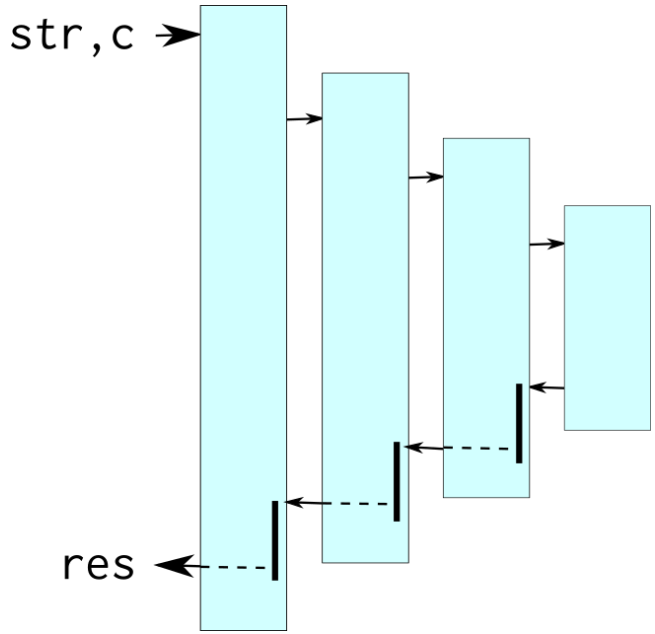
- **Abstraction** : la version (I) requiert une connaissance concrète de l'implémentation des chaînes de caractères, et de la façon d'itérer dessus ;
- **Indépendance** : les calculs de la version (F) n'interfèrent pas entre eux ; il n'y a pas d'état intermédiaire, explicite ou implicite.

Au final, le paradigme fonctionnel se présente comme encourageant la **composition** de transformations des valeurs. Les fonctions y jouent un rôle fondamental, ce qui donne le nom au paradigme.

Version impérative



Version fonctionnelle



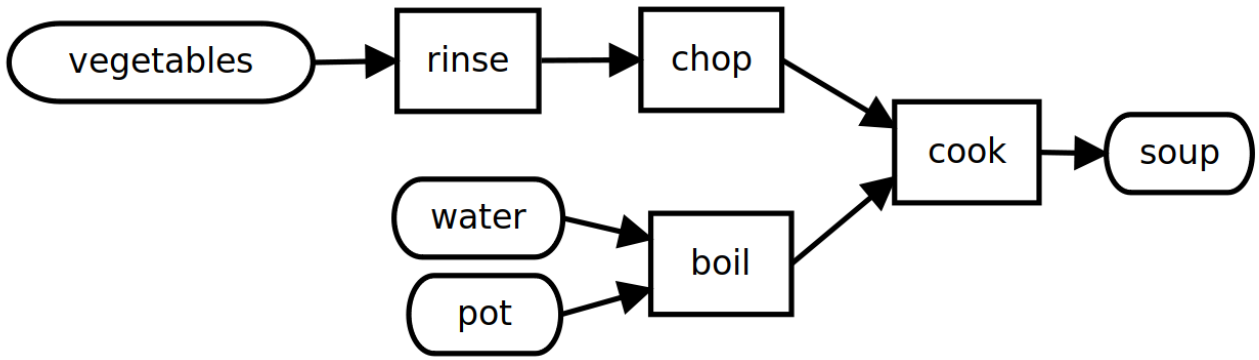
Aussi, le style fonctionnel promeut l'**indépendance** des calculs entre eux.

D'où l'étude de la programmation fonctionnelle. Elle se fait principalement sous deux angles particuliers :

- **Pureté, Modularité** : dans quelle mesure l'indépendance des calculs peut être mise à profit pour programmer, facilitant la réutilisation de code, la parallélisation, la reproductibilité, les preuves ...
- **Citoyenneté de 1ère classe** : dans quelle mesure la facilité de composition des fonctions induit des techniques de programmation variées : contrôle de l'évaluation, généralisations et spécialisation ...

※ Exercice : quel style utiliser ? (1/2)

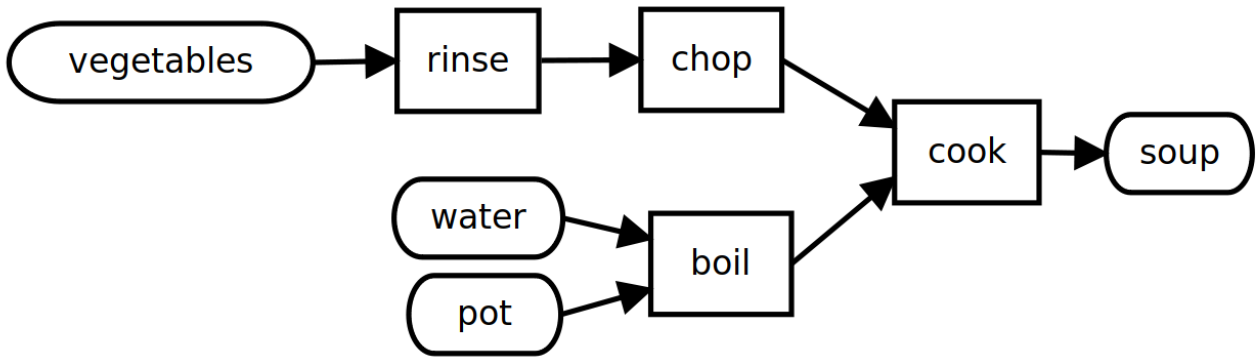
La description de l'algorithme suivante est elle impérative ou fonctionnelle ?



(adapté à partir d'une question posée sur Reddit /r/ProgrammingLanguages)

※ Exercice : quel style utiliser ? (1/2)

La description de l'algorithme suivante est elle impérative ou fonctionnelle ?



(adapté à partir d'une question posée sur Reddit /r/ProgrammingLanguages)

Réponse : **ni l'un ni l'autre**.

Il s'agit simplement d'une description des dépendances entre les calculs.
L'algorithme peut être implémenté de manière impérative ou fonctionnelle.
Néanmoins, une telle présentation encourage la composition des fonctions.

※ Exercice : quel style utiliser ? (2/2)

Version fonctionnelle

```
let vegetables = { type: "carrots", state: "raw" };
let water = { type: "water", volume: "1L" };
let pot = { contents: "empty" };

function rinse(v) {
  return { ...v, state: "rinsed" };
}
function chop(v) {
  if (v.state !== "rinsed") throw 'Unrinsed vegs';
  else return { ...v, state: "chopped" };
}
function boil(w, p) {
  if (p.contents !== "empty") throw 'Full pot';
  else return { ...p, contents: `hot ${w.type}` };
}
function cook(v, p) {
  if (v.state !== "chopped") throw 'Unchopped vegs';
  else
  if (p.contents !== "hot water") throw 'Cold pot';
  else return { ...p, contents: `soup` };
}
cook(
     chop(
          rinse(vegetables)
         ),
     boil(water,
          pot));      // → { contents: 'soup' }
```

Version impérative

```
let vegetables = { type: "carrots", state: "raw" };
let water = { type: "water", volume: "1L" };
let pot = { contents: "empty" };

function rinse(v) {
  v.state = "rinsed";
}
function chop(v) {
  if (v.state !== "rinsed") throw 'Unrinsed vegs';
  v.state = "chopped";
}
function boil(w, p) {
  if (p.contents !== "empty") throw 'Full pot';
  p.contents = `hot ${w.type}`;
}
function cook(v, p) {
  if (v.state !== "chopped") throw 'Unchopped vegs';
  if (p.contents !== "hot water") throw 'Cold pot';
  p.contents = `soup`;
}

rinse(vegetables);
chop(vegetables);
boil(water, pot);
cook(vegetables, pot);

pot;          // → { contents: 'soup' }
```

Une brève histoire de la programmation

La notion de programme est liée à la notion de **calculabilité** :

Quels calculs peut-on exprimer dans un langage de programmation ?

Une brève histoire de la programmation

La notion de programme est liée à la notion de **calculabilité** :

Quels calculs peut-on exprimer dans un ~~langage de programmation~~ ?
un modèle de calcul ?

Une brève histoire de la programmation

La notion de programme est liée à la notion de **calculabilité** :

Quels calculs peut-on exprimer dans un ~~langage de programmation~~ ?
un modèle de calcul ?

Au milieu des années 30 apparaissent successivement 3 modèles de calcul :

- le **lambda-calcul** proposé en 1936 par Alonzo Church,
- les **fonctions récursives** proposées en 1936 par Stephen Kleene,
- les **machines de Turing** proposées en 1937 par Alan Turing.

A quelques raffinements près, ces 3 modèles se trouvent être **équivalents** pour exprimer ces fameux calculs.

Il se trouve que ces modèles de calcul expriment différentes philosophies :

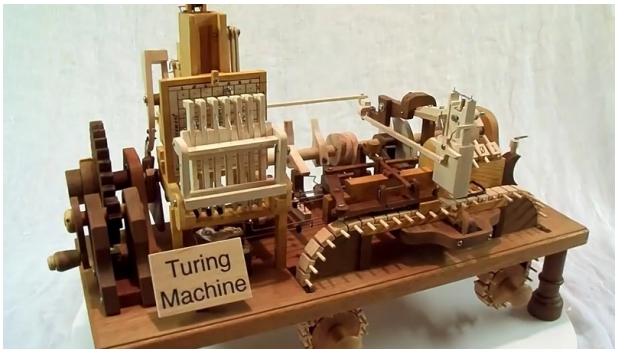
Le lambda-calcul

$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

$$\texttt{not true} \rightarrow_\beta \texttt{if true false true} \rightarrow_\beta \texttt{false}$$

- Un modèle formel “abstrait”
- Des fonctions mathématiques que l’on applique et compose

Les machines de Turing



Source : https://hackaday.com

- Une machine formelle “concrète”
- Un état global modifié par des instructions

Il se trouve que ces modèles de calcul expriment différentes philosophies :

Le lambda-calcul

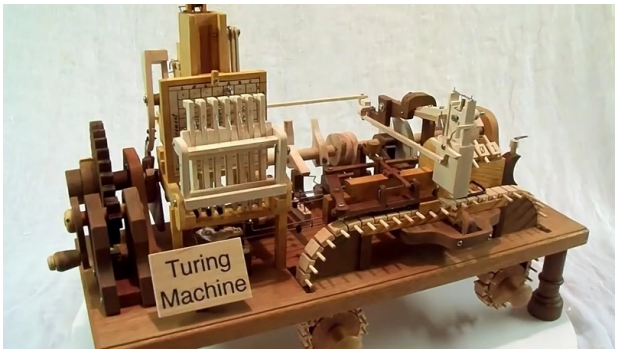
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

$$\texttt{not true} \rightarrow_\beta \texttt{if true false true} \rightarrow_\beta \texttt{false}$$

- Un modèle formel “abstrait”
- Des fonctions mathématiques que l’on applique et compose

“Style” fonctionnel

Les machines de Turing



Source : https://hackaday.com

- Une machine formelle “concrète”
- Un état global modifié par des instructions

“Style” impératif

Il se trouve que ces modèles de calcul expriment différentes philosophies :

Le lambda-calcul

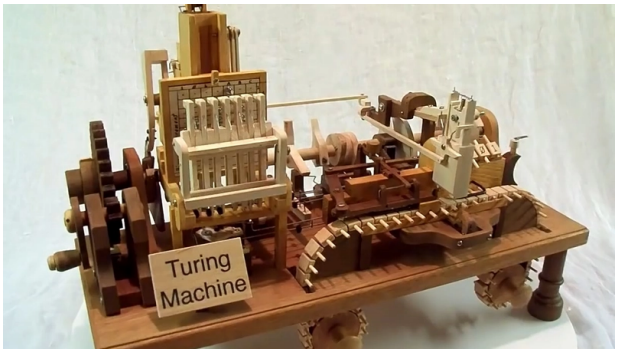
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

$$\texttt{not true} \rightarrow_\beta \texttt{if true false true} \rightarrow_\beta \texttt{false}$$

- Un modèle formel "abstrait"
- Des fonctions mathématiques que l'on applique et compose

"Style" fonctionnel

Les machines de Turing



Source : https://hackaday.com

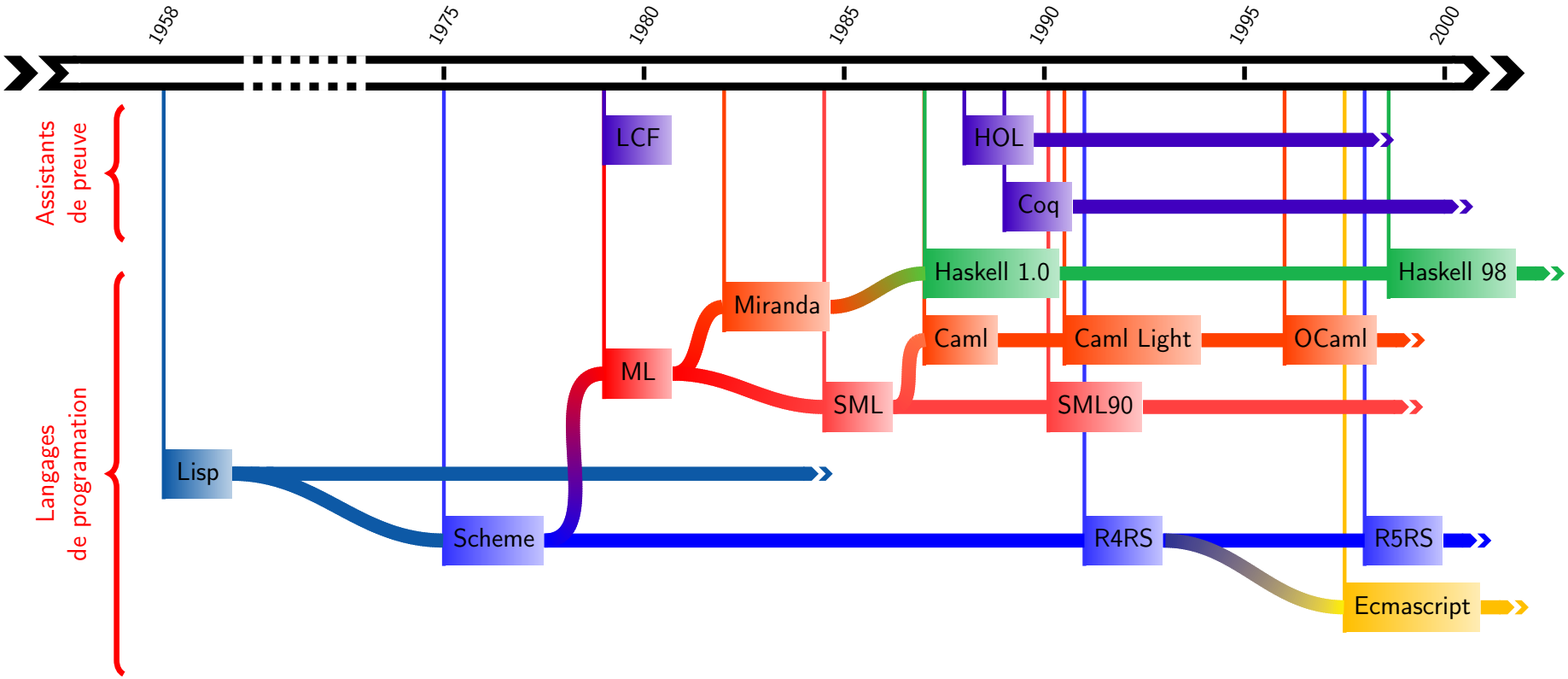
- Une machine formelle "concrète"
- Un état global modifié par des instructions

"Style" impératif

Ces philosophies ont énormément influencé (et influencent encore) les langages de programmation qui vinrent à la suite.

En particulier il existe une famille de langages fonctionnels, indissociables de domaines de recherche plus formels, comme la **vérification** et la preuve.

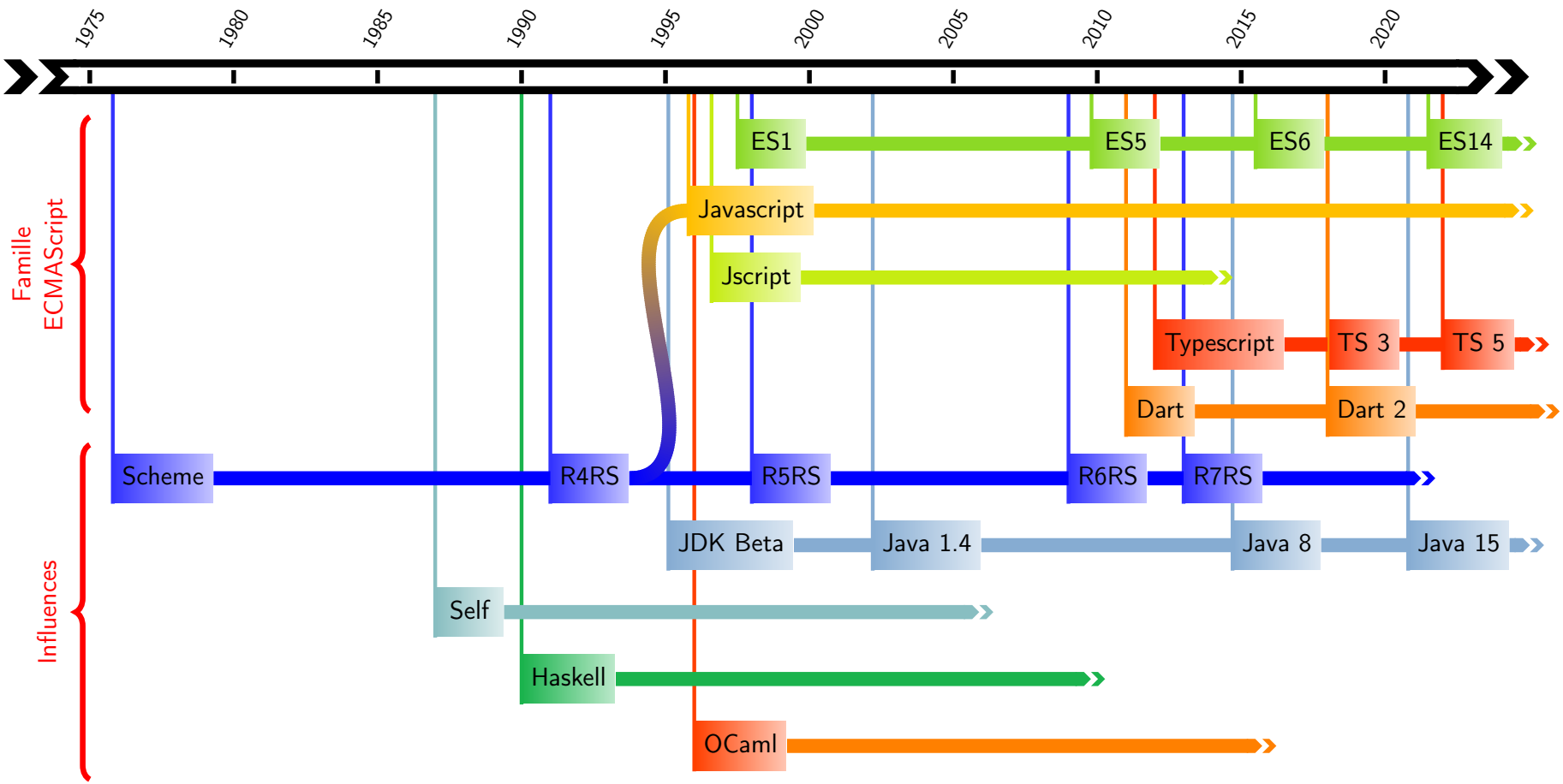
Une brève histoire des langages fonctionnels



Ecmascript

- Javascript est un langage de programmation créé en 1995 par Brendan Eich pour le navigateur Netscape.
- Influencé par Scheme (fonctionnel), Self (objet) et Java (syntaxe)
- Standardisé par l'ECMA International, à travers une spécification nommée **Ecmascript**.
- La version 6 de la norme (2015) est la modification majeure la plus récente, harmonisant les différentes implémentations précédentes.
- Le développement d'extensions au langage est incorporé chaque année dans une nouvelle révision de la norme.

Une brève histoire de Ecmascript



L'environnement Ecmascript

Les programmes Ecmascript s'utilisent de différentes manières :

- dans des navigateurs web, des scripts interagissent :
  - avec les pages HTML (à travers un arbre DOM),
  - avec les utilisateurs (à travers des événements),
  - avec des sites externes (avec par exemple Ajax ou Websocket)
- en tant que code autonome, exécuté par des moteurs comme Node.js,
- dans des bibliothèques de développement (Electron, Cordova), pour développer des clients lourds multi-plateformes.

L'environnement Node

L'environnement Node.js (`https://nodejs.org`) utilisé dans ce cours inclut :

- Le moteur d'exécution `node` qui sert aussi de REPL (acronyme de Read-Eval-Print Loop) ;
- Le gestionnaire de paquetages `npm` (`https://www.npmjs.com`) permettant d'installer des bibliothèques externes.

Exemples de bibliothèques encourageant la programmation fonctionnelle :

- `underscore` (`https://underscorejs.org`)
- `ramda` (`https://ramdajs.com`)
- `lodash` (`https://lodash.com`)

Philosophie de Ecmascript

- Inspiration objet forte et singulière

  (Langage à prototypes, les valeurs ont des méthodes . . .)

- Orientation dynamique très marquée

  (Typage dynamique, modification des méthodes des objets à la volée, introspection . . .)

- Tolérance aux erreurs considérable

  (Pas d'erreur si trop d'arguments, pas assez d'arguments, un champ manquant, un dépassement d'indice dans un tableau, pas d'erreurs arithmétiques)

Pour atténuer le peu de vérification dans le langage, Ecmascript 5 a introduit un **mode strict** qui transforme de nombreux comportements laxistes en erreurs.

Comparaison entre Ecmascript et le C

Le langage Ecmascript et le langage C, bien qu'éloignés, partagent des syntaxes très proches. Quelques différences notables de l'Ecmascript :

- Modèle d'exécution particulier

  (Interprétation par une machine virtuelle (e.g **V8**) générant du bytecode ou du code natif dans des phases de compilation "just-in-time")
- Typage dynamique

  (Pas d'indication de type devant les variables et les arguments, peu de vérification des types, nombreuses conversions implicites ...)
- Gestion de la mémoire automatique

  (Pas de `malloc` ou de `free`, allocation automatique, présence d'un ramasse-miettes)

... sans compter bien sûr tout ce qui a trait à la couche objet.

Ecmascript et la spécification

- La spécification du langage Ecmascript est accessible à :

  https://www.ecma-international.org/ecma-262

  Offre une référence complète, sinon digeste du langage.
- La documentation du langage par Mozilla est aussi une bonne source d'information :

  https://developer.mozilla.org/en-US/docs/Web/JavaScript

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

Ecmascript et la vérification

- Les langages Ecmascript souffrent d'un fort manque de vérification : types, nombre d'arguments, conversions implicites ...
- Plusieurs langages de programmation proposent d'ajouter un compilateur pour plus de vérification statique (Dart, Typescript ...)

Typescript (https://www.typescriptlang.org), un langage développé par Microsoft ajoute des types optionnels et un compilateur vérifiant ces types.

```
function double(n : number) : number {
    return 2*n;
}
console.log(double("six"));
```

```
% tsc file.ts
error: Argument of type 'string'
       is not assignable to
       parameter of type 'number'.
```

Il sera discuté pendant les TDs et utilisé pendant les projets.

La syntaxe du langage

- Cette partie décrit le langage et la syntaxe d'Ecmascript.
- Les éléments du langage sont présentés un par un de manière succinte.
  - les types primitifs,
  - les objets,
  - les tableaux,
  - les fonctions,
  - et les structures de contrôle.
- Pour des vétérans du C, cette syntaxe est très accessible.

Les valeurs

La spécification Ecmascript décrit 8 types de valeurs :

- `undefined` pour les valeurs non initialisées et `null` pour les valeurs nulles, vides ou inexistantes ;
- `boolean` pour les valeurs booléennes ; `true`, `false`
- `string` pour les chaînes de caractères immutables et `symbol` pour les types des clés des dictionnaires ; `"aster"`
- `number` et `bigint` pour les valeurs numériques ; `3.141`, `8642n`
- et les objets.

Parmi les objets, on trouve en particulier :

- les dictionnaires clé-valeurs ; `{ name: "yoda" }`
- les tableaux ; `[ "un", "dos" ]`
- les fonctions. `function (n) return n+1;`

Les types primitifs en Ecmascript (1/2)

Définition (Type primitifs)

Les types primitifs sont tous les types de valeurs non-objets :

`boolean`, `string`, `symbol`, `number`, `bigint`

- Les types primitifs sont tous **immutables**.
- Les valeurs associées ne sont pas des objets (même si la syntaxe leur en donne l'apparence).
- Elles possèdent des **propriétés** et des **méthodes**.

Les types primitifs en Ecmascript (2/2)

Exemple : les chaînes de caractères

Toute valeur de type `string` possède :

- une propriété `length` ;
- 32 méthodes (dont `concat`, `match`, `replace`, `search`, `substring` ...)

```
let s = "archeopteryx";
s[0];                 // → "a" (returns a string)
s[0] = "z";           // no error
s;                    // → "archeopteryx" (s is unchanged since it is immutable)

s.length;             // → 12
s.substring(2, 7);    // → "cheop"
s.match(/c.*p/);      // → [ 'cheop', index: 2 (...) ]
```

En 1ère approximation, une méthode est une fonction prenant un paramètre privilégié et utilisant une syntaxe particulière.

Les objets en Ecmascript (1/2)

- Les objets sont, en 1ère approximation, des **dictionnaires** clé-valeur.

```
let o = {};          // → { }
o.firstName = "Abe";
o.lastName  = "Lincoln";
o;                   // → { firstName: 'Abe', lastName: 'Lincoln' }
o['firstName'];      // → "Abe"
```

- Les objets sont **mutables** et **itérables**.

```
o.firstName = "Abraham";
o;                   // → { firstName: 'Abraham', lastName: 'Lincoln' }
Object.keys(o);      // → [ 'firstName', 'lastName' ]
Object.keys(o).forEach( (p) => { o[p] = o[p] + "␣(updated)"; });
o; // { firstName: 'Abraham (updated)', lastName: 'Lincoln (updated)' }
```

Les objets en Ecmascript (2/2)

- Chaque objet possède une propriété particulière : son **prototype**.
- Le prototype contient l'ensemble des **méthodes** associées à cet objet :

```
let o = {};
o.__proto__;  // → [Object: null prototype] {}
Object.getOwnPropertyNames(o.__proto__); //→[.., 'toString', 'valueOf', ..]
```

- Le prototype est modifiable à volonté :

```
String.prototype.halfLength = function () {
  return Math.floor(this.length / 2); };
new String("abraracourcix").halfLength(); // → 6
```

- Une variable particulière (`this`) fait référence à l'objet courant.

Les tableaux

- Les tableaux sont des objets particuliers.
- Les clés de ces objets sont les indices utilisés du tableau.

```
const fruits = ["Apple", "Banana"]; // only the reference is const
fruits[0];      // → 'Apple'
fruits.length; // → 2
```

- Rien n'oblige les indices à être consécutifs.
  Rien n'oblige les éléments à être du même type.

```
fruits[4] = "Coconut";
fruits;         // [ 'Apple', 'Banana', <2 empty items>, 'Coconut' ]
fruits[2] = 666;
fruits;         // [ 'Apple', 'Banana', 666, <1 empty item>, 'Coconut' ]
```

Les fonctions

- Les fonctions sont des objets particuliers.
- Il existe deux manières différentes de les construire :
  - les fonctions classiques (*regular functions*)

    Version nommée :

    ```
    function s(x) { return x+1; }
    ```

    Version **anonyme** :

    ```
    let s = function (y) { return y-1; }
    ```

  - les fonctions fléchées (*arrow functions*)

    Version standard :

    ```
    (x) ⇒ { return x*2; }
    ```

    Version avec retour implicite :

    ```
    (y) ⇒ 3*y;
    ```

- Les différences entre les 2 constructions sont mineures dans ce cours.

Par exemple, les fonctions classiques peuvent accéder à une variable `arguments` :

```
function dispArgs() { return arguments; }
dispArgs(1, 2, "trois"); // → [Arguments] { '0': 1, '1': 2, '2': 'trois' }
```

De l'importance des blocs

Définition (Bloc)

Un **bloc** est une construction syntaxique particulière délimitant une zone contigüe de code (typiquement entre deux accolades).

```
function aFunction(a, b, c) {
  if (b*b - 4*a*c >= 0) {
    return "real";
  } else {
    return "complex";
} }
```

Bloc "if-true"

```
let anArr = [1,2,3,4,5];
for (let i=0; i<anArr.length; i++) {
  let j = i*i;
  console.log(anArr[i] + j);
}
console.log(anArr);
```

Bloc "for-loop"

Les fonctions acquièrent ainsi un statut spécial : ce sont des blocs **nommés** (donc transportables et réutilisables) et **paramétrés**.

Structures de contrôle

La syntaxe Ecmascript inclut les structures de contrôle suivantes :

- Les expressions conditionnelles :

```
if ⟨bool⟩ {
  ⟨block-true⟩
} else {
  ⟨block-false⟩
}
```

- Les aiguillages (*switch*) :

```
switch ⟨expr⟩ {
case ⟨val⟩:
  ⟨code-ending-with-break⟩
default:
  ⟨code-dflt⟩
}
```

- Les boucles bornées :

```
for ( ⟨init⟩; ⟨test⟩; ⟨next⟩ ) {
  ⟨block-loop⟩
}
```

- Les boucles non bornées :

```
while ( ⟨test⟩ ) {
  ⟨block-loop⟩
}
```

Noter la décomposition en blocs variant selon la structure de contrôle.

Portée des données

Considérons un programme et réfléchissons à la question suivante :

> Qui a accès à quoi ?

- Notion de visibilité, de portée, ou de droits d'accès ;
- Mécanismes multiples selon les langages de programmation ;

Une gestion soignée des visibilités permet de nombreuses techniques logicielles, généralement basées sur la notion d'**abstraction**.

Définition (Abstraction)

Propriété logicielle selon laquelle les composants séparent leur part publique (interface) de leur part privée (implémentation).

Variables et liaisons

Définition (Variable)

Une **variable** est une association entre un nom (son identifiant) et un emplacement stockant une valeur lors de l'exécution (sa valeur).

Pour créer une variable, il suffit de la déclarer :

```
let aVar = "X"; // example of variable named 'aVar' and storing the value "X"
```

- Une variable est une **liaison** (*binding*) entre un nom et une valeur.
- Le mot "variable" est parfois mal venu, la liaison pouvant être constante.
- A fur et à mesure de l'exécution, l'ensemble des variables et les valeurs qu'elles contiennent constituent un **environnement**.

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Portée lexicale de *t*

Portée (lexicale)

Définition (Portée)

La **portée** (*scope*) d'un identifiant dans un code donné est la région du code dans laquelle la variable associée est accessible dans l'environnement.

La **portée lexicale** (*lexical scope*) d'un identifiant débute au moment de sa déclaration et se termine à la fin du bloc dans lequel il est défini.

```
function foo(t) {
  return 2*t;
}
function baz(z) {
  console.log(z);
  let t = 42;
  return foo(z+t);
}
baz(10);
```

Une autre portée

Portée lexicale de *t*

Remarque

Si deux identifiants sont les mêmes, celui dans le bloc courant **masque** l'autre (*variable shadowing*).

La norme Ecmascript 6 propose deux manières de déclarer des variables avec une portée **lexicale** :

- `let` pour une variable (la liaison dans l'environnement est mutable)
- `const` pour une constante (la liaison dans l'environnement est constante)

Le langage autorise aussi de déclarer des variables avec la syntaxe suivante :

- `var` pour une variable à la portée non lexicale

Mais ceci permet entre autres d'utiliser une variable **avant** sa déclaration.

Principe de localité des variables

Dans ce cours, et afin de mieux contrôler les visibilités, on se limitera à n'utiliser des variables qu'avec une portée **lexicale**.

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;
function foo() {
  let h = 2;
}
function baz() {
  let i = 3;
  foo();
}
baz();
```

Une boucle dans une fonction

```
let g = 1;
function loop() {
  let h = 2;
  for (let i=3; i<4; i++) {
    console.log(g+h+i);
  }
}
loop();
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;
function loop() {
  let h = 2;
  for (let i=3; i<4; i++) {
    console.log(g+h+i);
  }
}
loop();
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;
function foo() {
  let g = 2;
  function baz() {
    let g = 3;
  }
  baz();
}
foo();
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;              // g_1
function foo() {
  let g = 2;            // g_2
  function baz() {
    let g = 3;          // g_3
  }
  baz();                // g_2
}
foo();                  // g_1
```

Une fonction renvoyant une fonction

```
function foo() {
  let h = [];
  function baz() {
    h.push(1);
    return h;
  }
  return baz;       // not a typo
}
let f = foo();
f();
```

Exemples de portées lexicales

Deux fonctions indépendantes

```
let g = 1;              // g
function foo() {        // g
  let h = 2;            // g, h
}
function baz() {        // g
  let i = 3;            // g, i
  foo();
}
baz();                  // g
```

Une boucle dans une fonction

```
let g = 1;                          // g
function loop() {                   // g
  let h = 2;                        // g, h
  for (let i=3; i<4; i++) {         // g, h, i
    console.log(g+h+i);             // g, h, i
  }
}
loop();                             // g
```

Une liaison masquant une autre liaison

```
let g = 1;              // g_1
function foo() {
  let g = 2;            // g_2
  function baz() {
    let g = 3;          // g_3
  }
  baz();                // g_2
}
foo();                  // g_1
```

Une fonction renvoyant une fonction

```
function foo() {    //
  let h = [];       // h (an array)
  function baz() { // h
    h.push(1);      // h
    return h;       // h
  }
  return baz;       // h
}
let f = foo();      // f
f();                // f (returns h = [1])
```

Portée

A titre culturel, noter qu'il existe en programmation (et en Ecmascript) d'autres formes de portée des variables :

- la portée **globale** — L'identifiant est accessible à partir de son point de définition et dans toute la suite du code.
- la portée **dynamique** — L'identifiant possède une pile de liaisons différentes qui évolue au cours de l'exécution.
- la portée **fonction** — L'identifiant déclaré au milieu d'une fonction est accessible dans toute la fonction, même au début.

Même si ces notions ont chacun leur histoire et leurs avantages (et possiblement inconvénients), elles sortent du cadre de ce cours.

Durée de vie

Autre question naturelle concernant les données dans un programme :

Qui vit combien de temps ?

Définition (Durée de vie)

La **durée de vie** (*lifetime*) d'une valeur est l'intervalle de temps pendant l'exécution du programme où cette valeur est accessible.

Gestion de la durée de vie

La durée de vie des valeurs peut-être gérée de différentes manières :

- En les plaçant dans la **pile** (*stack*), la gestion est **automatique** : elles terminent leur vie au dépilement ou survivent par copie.

  Ex : allocation par défaut en C, types primitifs en Ecmascript (V8)

- En les plaçant dans le **tas** (*heap*), la gestion peut alors se faire :
  - en allouant et désallouant la mémoire de manière **manuelle**,

    Ex : allocation dynamique en C avec `malloc` et `free`

  - ou en délaissant cette gestion à un système **automatique** comme un **ramasse-miettes** (*garbage collector*).

    Ex : allocation des objets en Ecmascript (V8), gérés comme des références

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

```
function setEmpty() {
  return {};
}
function setSize(aSet) {
  return Object.keys(aSet).length;
}
function setAdd(aSet, anElem) {
  aSet[anElem] = 1; // dummy val
}
```

Exemple de durée de vie

La fonction `countUnique` compte le nombre d'éléments distincts dans un tableau de nombres :

```
function countUnique(anArray) {
  let aSet = setEmpty();
  anArray.forEach((anElem) ⇒ {
      setAdd(aSet, anElem);
  });
  return setSize(aSet);
}
countUnique([1,2,3,1,2,3]); //→ 3
```

```
function setEmpty() {
  return {};
}
function setSize(aSet) {
  return Object.keys(aSet).length;
}
function setAdd(aSet, anElem) {
  aSet[anElem] = 1; // dummy val
}
```

La variable stockée dans `aSet` est :

- allouée dans `setEmpty`,
- est transmise et mise à jour lors des appels à `setAdd` et `setSize`,
- et finalement détruite au retour de `countUnique`.

Exemple de manipulation des portées (1/2)

Considérons l'exemple d'un générateur pseudo-aléatoire (Lehmer) :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

Si on répète la fonction `rand()` 10 fois, on obtient :

8357, 36942, 18096, 46460, 11039, 41481, 30836, 18905, 41598, 39611

Problèmes :
- la variable `seed` est globale, tout le code peut la modifier ;
- les constantes du générateur sont publiques.

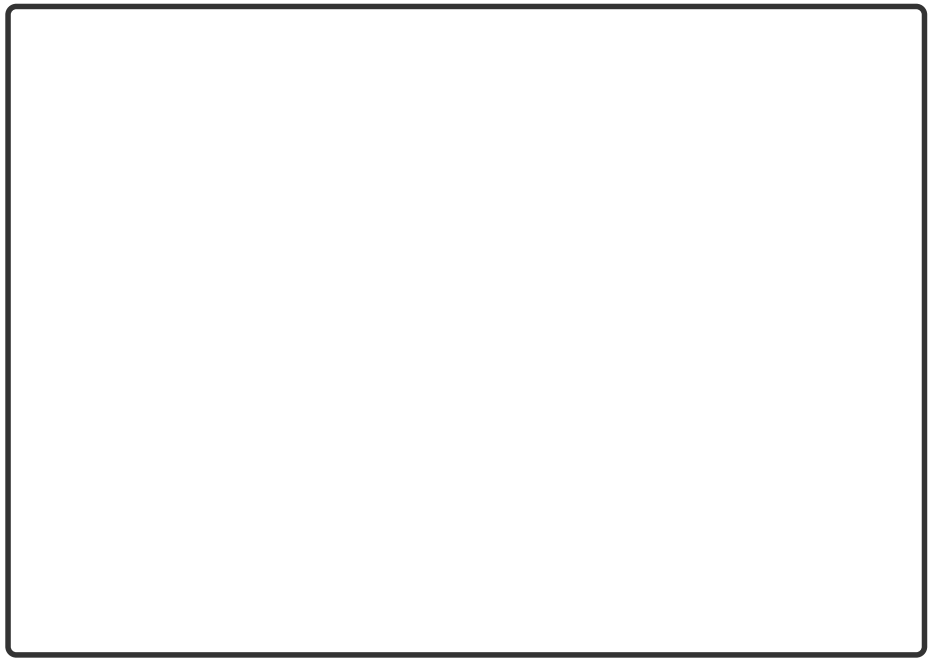
Solution :
- limiter les portées en encapsulant le code dans un bloc.

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$



Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$

```
{
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };

}
```

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

$\Rightarrow$

```
function makeRng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };

}
```

Exemple de manipulation des portées (2/2)

Encapsulons les paramètres, tout en laissant la fonction `rand` accessible :

```
let seed = 12345;
const m = 65537; const a = 75;
function rand() {
  seed = (a * seed) % m;
  return seed;
};
```

⇒

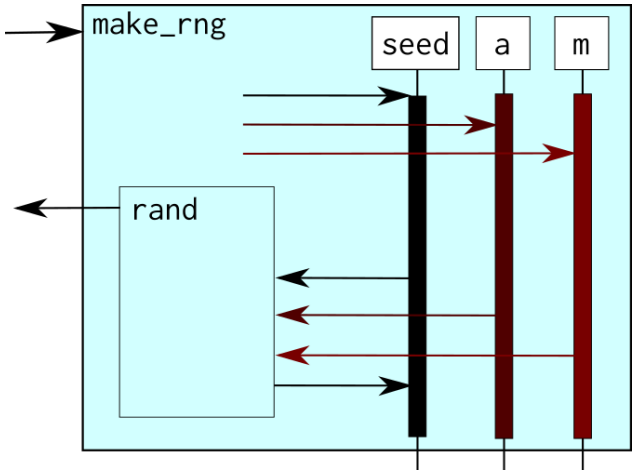
```
function makeRng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };
  return rand;
}
let rand = makeRng();
```

Les variables `seed`, `a` et `m` sont locales à la fonction `makeRng`.
Néanmoins, elles restent **accessibles** à la fonction `rand`, hors de `makeRng`.

Fermetures

Définition (Fermeture)

Une **fermeture** (*closure*) consiste en la donnée d'une fonction, ainsi que de l'environnement nécessaire à son fonctionnement.



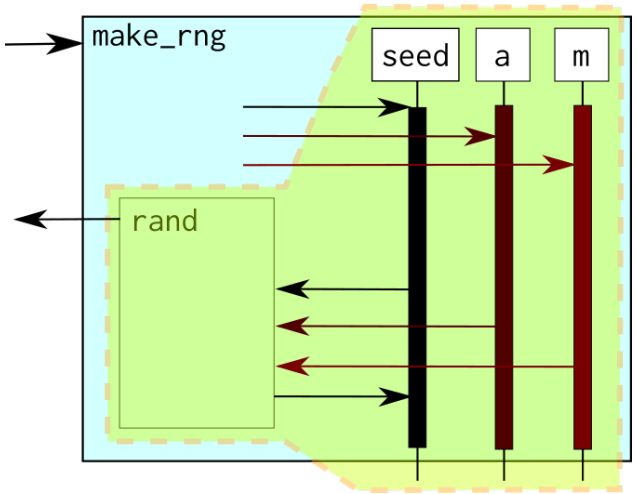
- Grâce aux fermetures, les fonctions deviennent des entités autonomes.
- Grâce aux portées, les fonctions ont une séparation public/privé.

Ces propriétés font des fonctions des petites unités de code portables.

Fermetures

Définition (Fermeture)

Une **fermeture** (*closure*) consiste en la donnée d'une fonction, ainsi que de l'environnement nécessaire à son fonctionnement.



- Grâce aux fermetures, les fonctions deviennent des entités autonomes.
- Grâce aux portées, les fonctions ont une séparation public/privé.

Ces propriétés font des fonctions des petites unités de code portables.

Conclusion sur la vie des données

- Un des aspects importants de la programmation consiste à bien comprendre les **dépendances** entre les entités d'un code.
- La notion de **portée** (et d'accessibilité) est une notion centrale pour comprendre les dépendances.
- Les fonctions, à travers les mécanismes de fermeture et de portée, permettent de contrôler ces dépendances.
- En cela, elles permettent un style de programmation basé sur l'indépendance des calculs.

Prochain chapitre, la **pureté**, une qualité pour limiter les dépendances.

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

- Le `if-then-else` est une expression* ;
- Il s'agit d'un style **fonctionnel**, proche de la définition mathématique ;
- Le code est explicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur pourrait utiliser cette information pour **appliquer des optimisations**.

※ Exercice : quel style utiliser ? (1/2)

Comparons les deux écritures d'une fonction très simple :

```
function fact(n) {
  if (n <= 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
function fact(n) {
  if (n <= 1)
    return 1; // Early return
  // ←---------- Nothing here !
  return n * fact(n-1);
}
```

Quel style adopter ?

- Le `if-then-else` est une expression* ;
- Il s'agit d'un style **fonctionnel**, proche de la définition mathématique ;
- Le code est explicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur pourrait utiliser cette information pour **appliquer des optimisations**.

- Le `if-then` est une instruction ;
- Il s'agit d'un style **impératif**, parfois aussi appelé un **early return** ;
- Le code est implicite sur le fait que les deux cas forment une disjonction ;
- Un compilateur sera gêné pour optimiser ;
- La fonction est **plus lisible**.

Il n'y a pas de réponse évidente.

※ Exercice : quel style utiliser ? (2/2)

Quelles types d'optimisations un compilateur pourrait-il appliquer ?
⇒ principalement des vérifications par **analyse statique**.

Exemple : les discriminated unions en Typescript

```typescript
interface Circle { kind: "circle"; radius: number; }
interface Square { kind: "square"; sideLength: number; }

type Shape = Circle | Square;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
  }
}
```

Un `Circle` a un attribut `radius` contrairement à un `Square`.

Un `Shape` est soit un `Circle`, soit un `Square`.

Dans ce bloc, le compilateur sait que `shape` possède un champ `radius` contrairement au bloc suivant.

Il peut le faire car la distinction entre les deux cas est explicite.

Cf. le **pattern matching** des langages comme OCaml et Haskell.

Tests d'égalité et vérification

En Ecmascript, il existe plusieurs façons de tester l'égalité entre 2 valeurs :

- Les opérateurs d'**égalité faible** `==` et `!=` :

```
"0" == 0;     // → true
0   == false; // → true
```

Conversion des éléments à comparer avant d'effectuer la comparaison.

- Les opérateurs d'**égalité stricte** `===` et `!==` :

```
"0" === 0;     // → false
0   === false; // → false
```

Même comparaison mais **sans conversion** préalable.
En particulier, la comparaison renvoie `false` si les types sont différents.

Règle (utilisation de comparaisons plus sûres)

Systématiquement utiliser les opérateurs d'égalité stricte.

Interpolation de chaînes de caractères

En Ecmascript, un moyen pratique d'engendrer des chaînes de caractères :

```
let aNumber = 6;
let aString = "Rastapopoulos";
// The next string is surrounded with backquotes ‘‘
console.log(‘The␣devious␣${aString}␣was␣teasing␣agent␣00${aNumber}‘);
// Log : The devious Rastapopoulos was teasing agent 006
```

Il s'agit d'une **interpolation** (*template literals*), faite en :

- Délimitant la chaîne par des backquotes `‘` (AltGr + 7 sur un clavier AZERTY).
- Encapsulant les valeurs à insérer dans des blocs `${}`.

Règle (lisibilité des chaînes de caractères)

Privilégier les interpolations pour construire des chaînes complexes.

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`

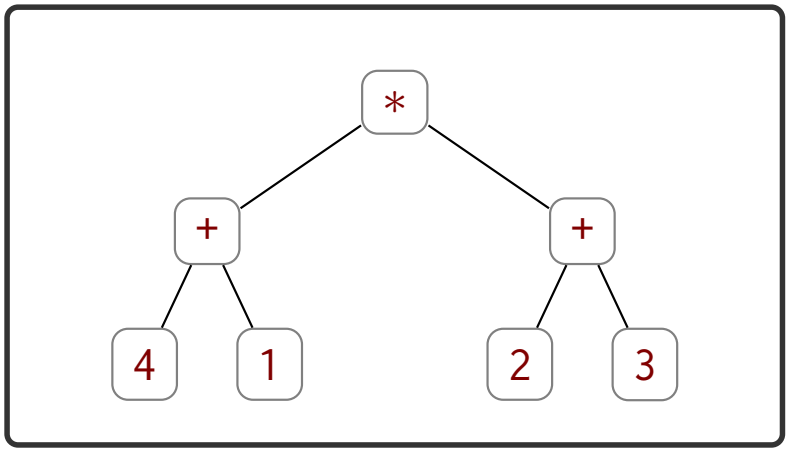
```
Math.random(); // → a number between 0 and 1
```

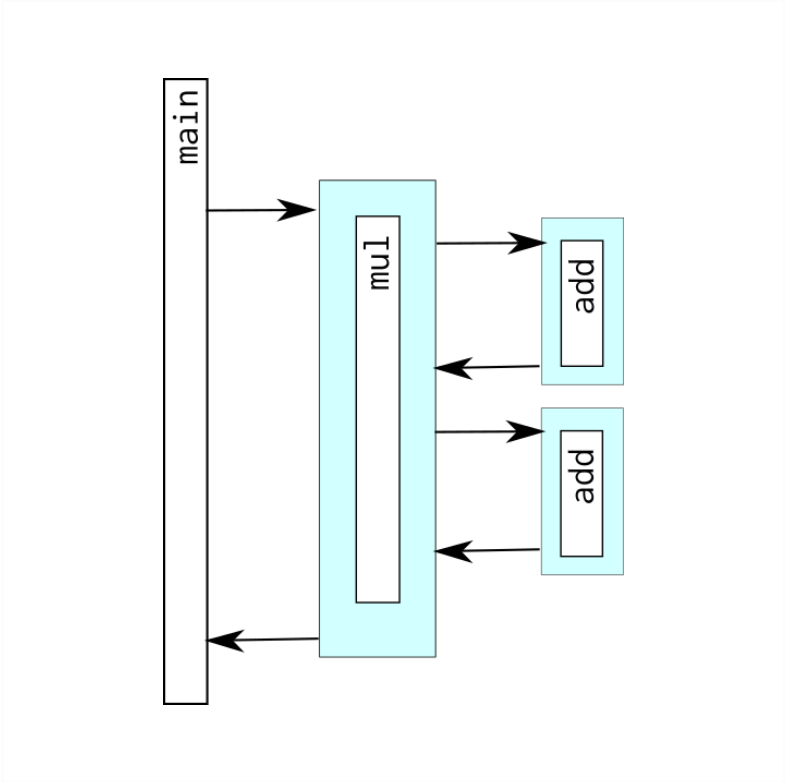
- `Date.now`

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

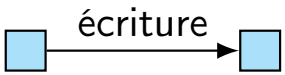
Cette différence de philosophie se traduit donc dans les programmes.

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



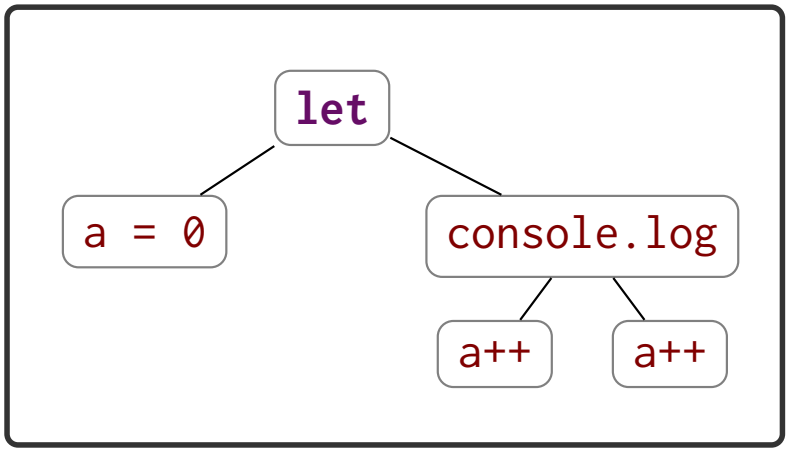


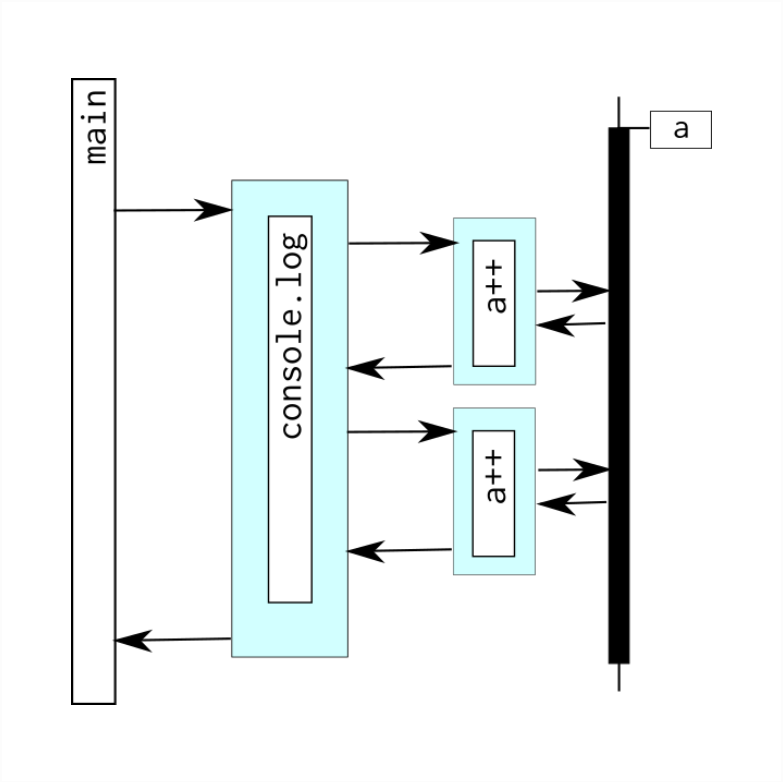
Les flèches indiquent les **dépendances** que les calculs entretiennent entre eux.



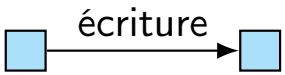
Cette différence de philosophie se traduit donc dans les programmes.

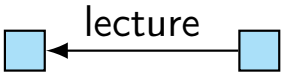
```
function main() {
  let a = 0;
  console.log(a++, a++);
}
```





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);
}
```

⇒

```
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**.

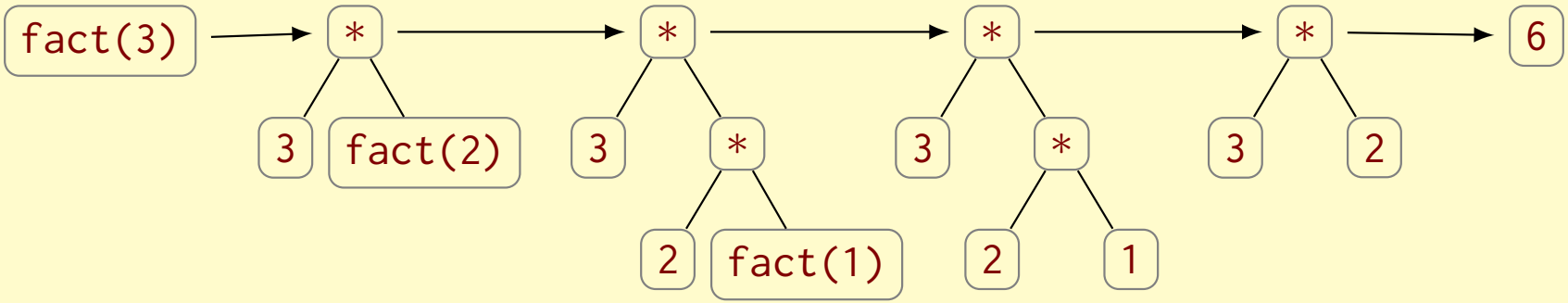
Définition (Récursif)

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.

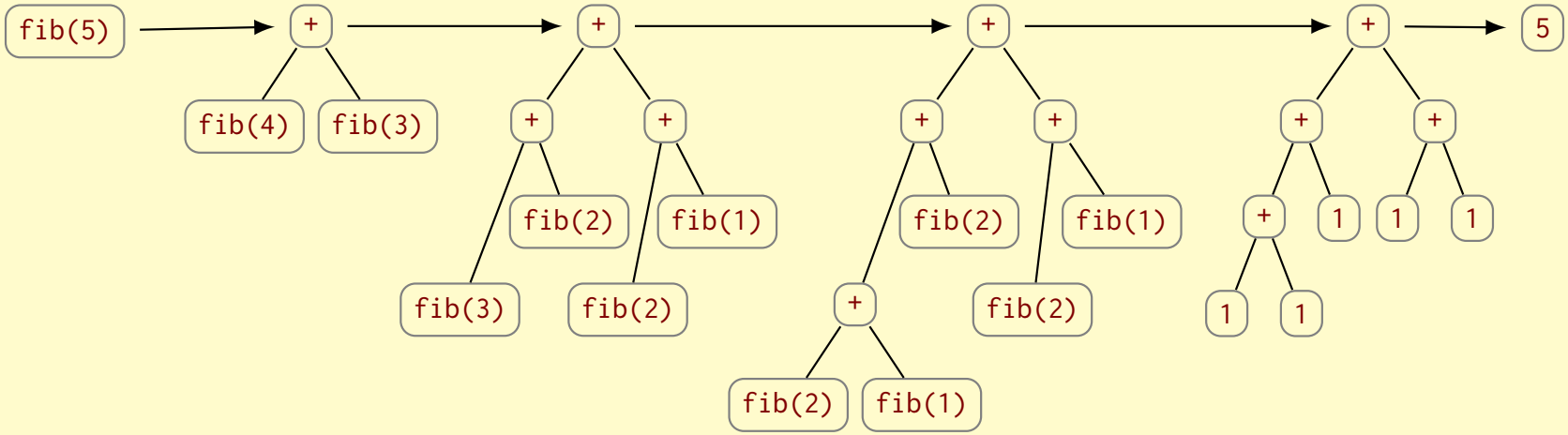
Limites des algorithmes récursifs

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” :

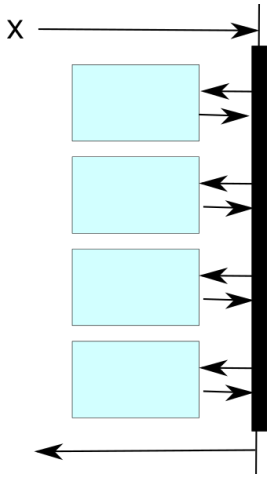
- La boucle classique :

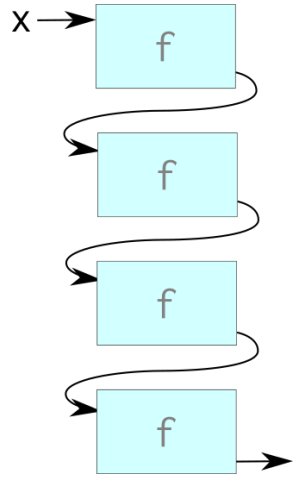
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

- 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 ...

♪ Les booléens, des valeurs comme les autres

En Ecmascript, les booléens sont représentés par les valeurs `true` et `false`.
Mais le langage effectue de nombreux types de conversions implicites :

```
true + true;   // → 2
false * false; // → 0
```

```
1 ? "one" : "two"; // → "one"
0 ? "one" : "two"; // → "two"
```

Les opérateurs booléens ont des règles d'évaluation particulières :

```
function log(s, b) { console.log(s); return b; }
log("left", true)  || log ("right", true); // → true,  Log : "left"
log("left", false) && log ("right", true); // → false, Log : "left"
```

Règle (utilisation des opérateurs sur leurs types naturels)

Privilégier l'utilisation des opérateurs sur des valeurs du type correspondant (e.g `!` s'utilise sur des `bool`, `+` s'utilise sur des `number`).

La reconnaissance de motifs

Ecmascript définit une sorte de **reconnaissance de motifs**.
Ainsi, il est possible de définir ou d'affecter des variables ainsi :

```
[a, b, c = 30]  = [10, 20];                  // a = 10, b = 20, c = 30
[a, b]          = [b, a];                    // a = 20, b = 10
[a, b, ...rest] = [10, 20, 30, 40, 50];      // a = 10, b = 20, rest = [30,40,50]
```

```
({ k1: a, k2: b, k3: c = "c"} = { k1:"a", k2:"b" });  // a='a',b='b',c='c'
({ k1: a, ...rest }    = { k1:"a", k2:"b", k3:"c" }); // a='a',rest={k2:'b',k3:'c'}
```

Exemples d'application

- Renvoyer plusieurs variables ;
- Décomposer aisément un objet complexe ;
- Définir des arguments par défauts pour une fonction.

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.

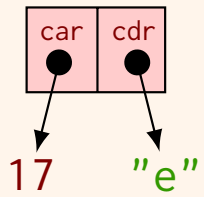
- Nommés aussi types de données algébriques, types sommes ou variants.

```
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`.



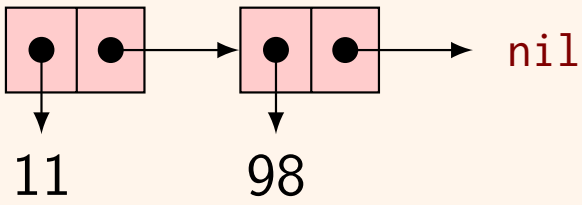

```
{ car: 17, cdr: "e" }
```

```
// 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.

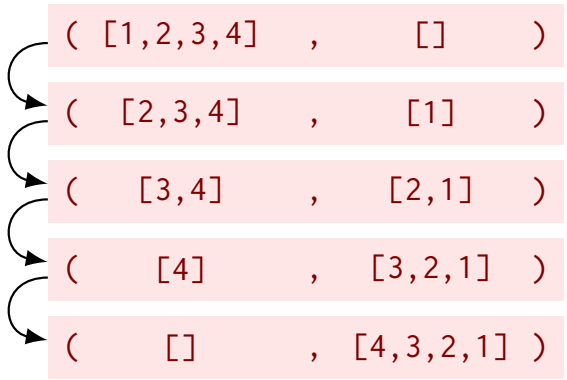
Influence de la structure inductive

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
}
```

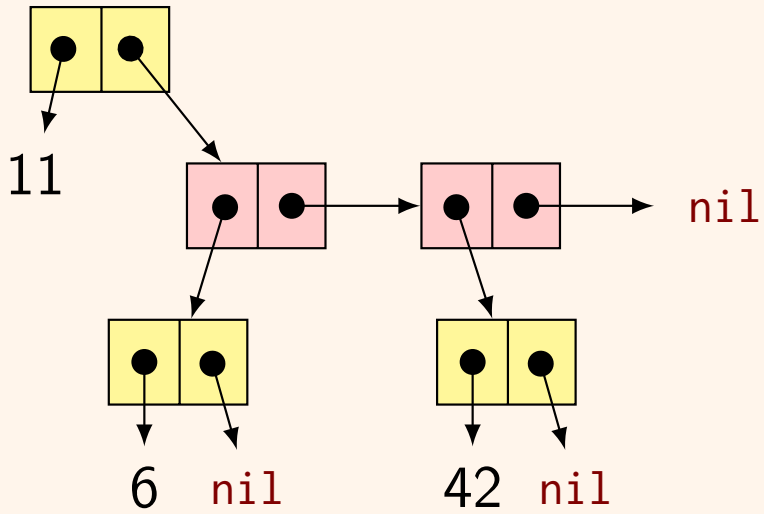
Parmi les caractéristiques mises en avant :

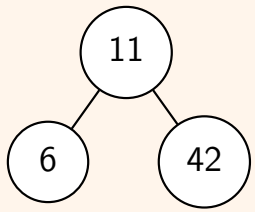
- les listes sont immutables, et donc adaptées à la programmation pure ;
- les listes sont immutables, permettant des optimisations comparables à ce que l'on peut obtenir avec des tableaux.

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)` $\longrightarrow$ `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*). 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.

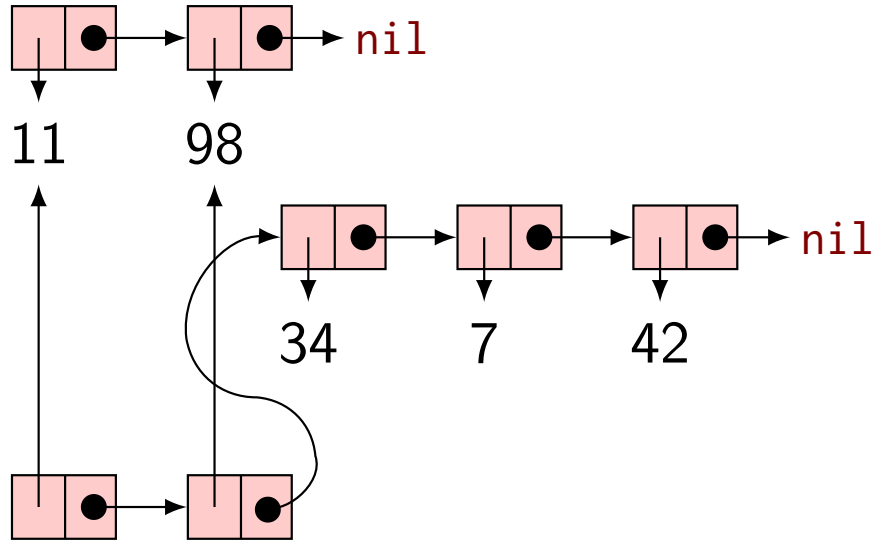
Fait

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;
}
```

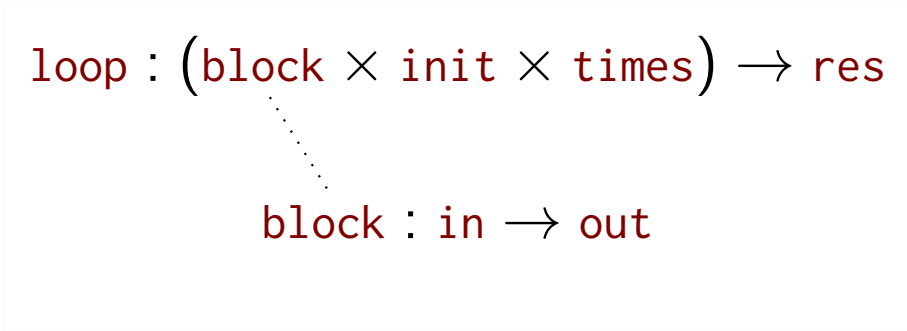
- `loop` est une fonction qui prend en paramètre une fonction `block`.
  Il s'agit d'une **fonction d'ordre supérieur**.
- Décrit un algorithme complexe à partir d'un algorithme plus simple.

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 :

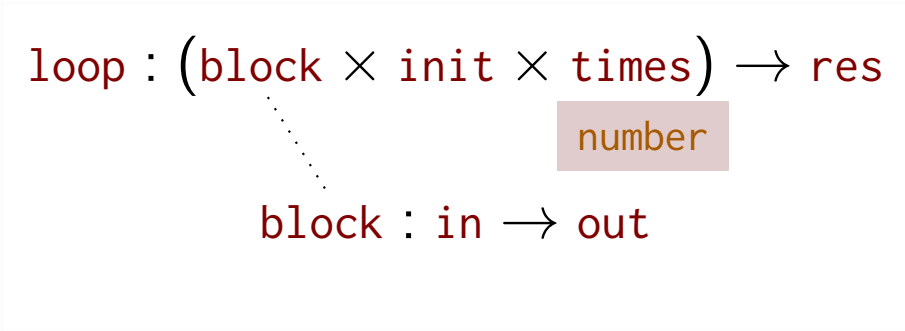
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



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 :

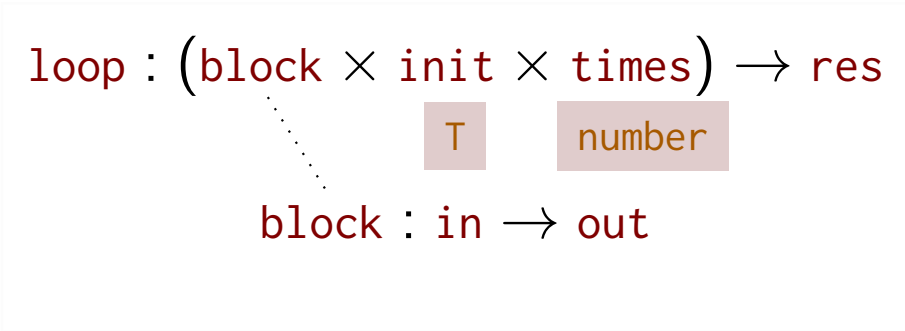
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



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 :

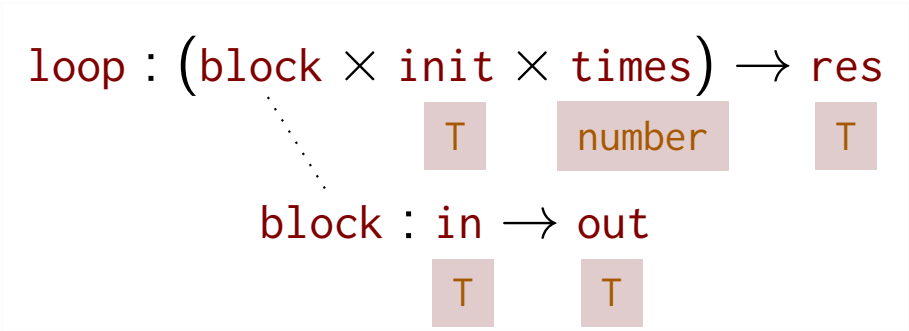
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



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 :

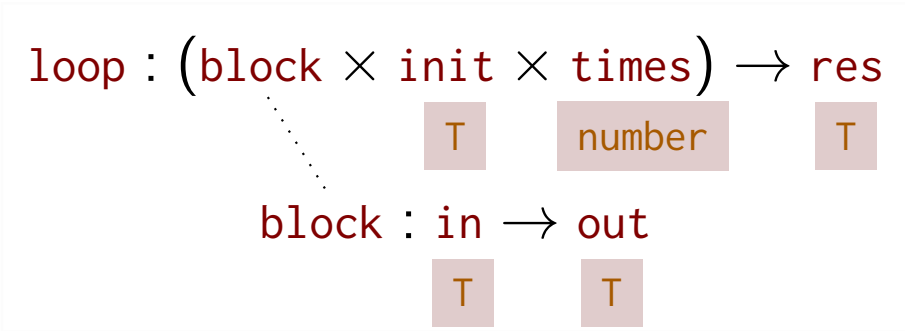
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```



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

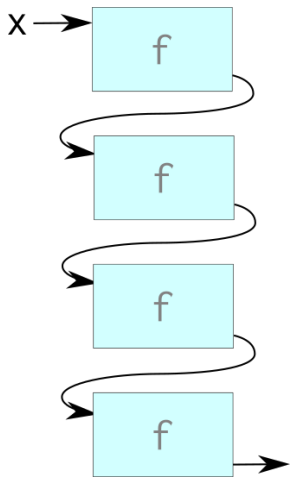
```
function filter<T>(block: (T) ⇒ boolean, arr: T[]) : T[] {
    const brr = [];
    for (let i = 0, j = 0; i < arr.length; i++) {
        if (block(arr[i]))
            brr[j++] = arr[i];
    }
    return brr;
}

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)

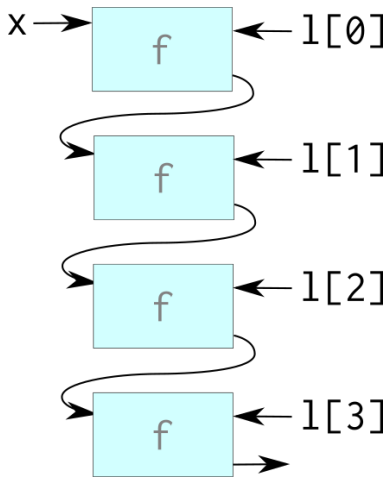
Quelle est la version fonctionnelle pure d'un **parcours sur une liste** `l` ?
Pour rappel, nous avions établi le schéma suivant pour une simple boucle :




```
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)

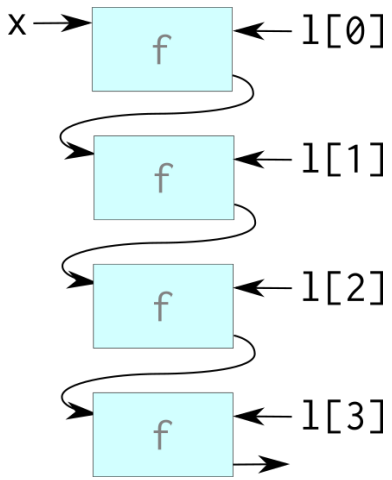
Quelle est la version fonctionnelle pure d'un **parcours sur une liste** `l` ?
Une idée simple consiste à passer les éléments de la liste `l` dans la boucle :




```
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*).

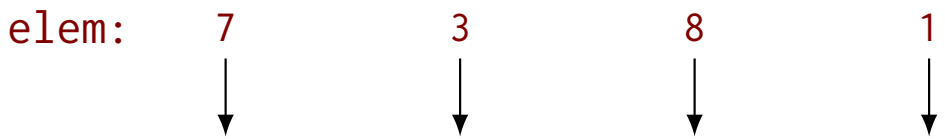
Ordre supérieur : les pliages (2/3)

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 :



`acc: 0`

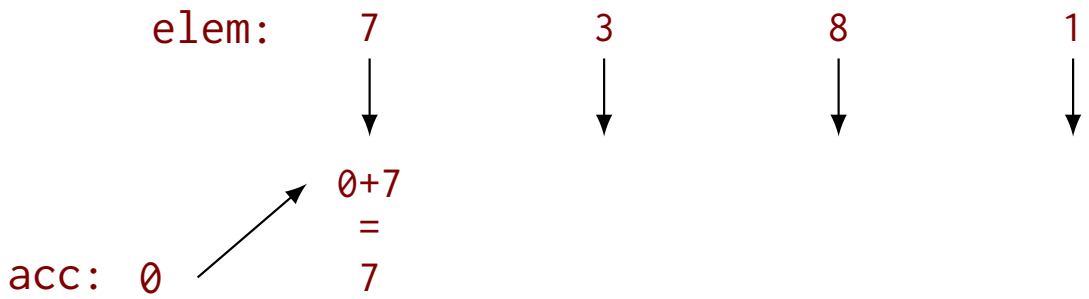
Ordre supérieur : les pliages (2/3)

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 :



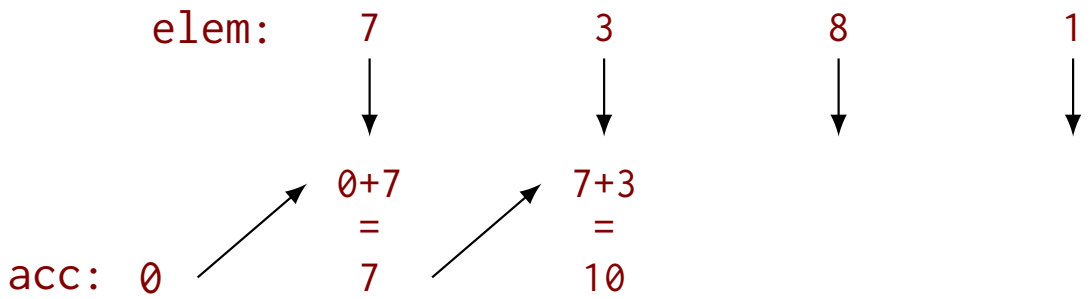
Ordre supérieur : les pliages (2/3)

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 :



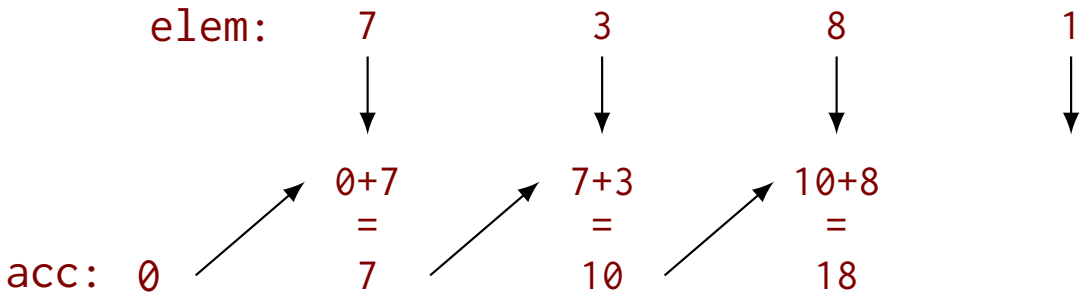
Ordre supérieur : les pliages (2/3)

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 :



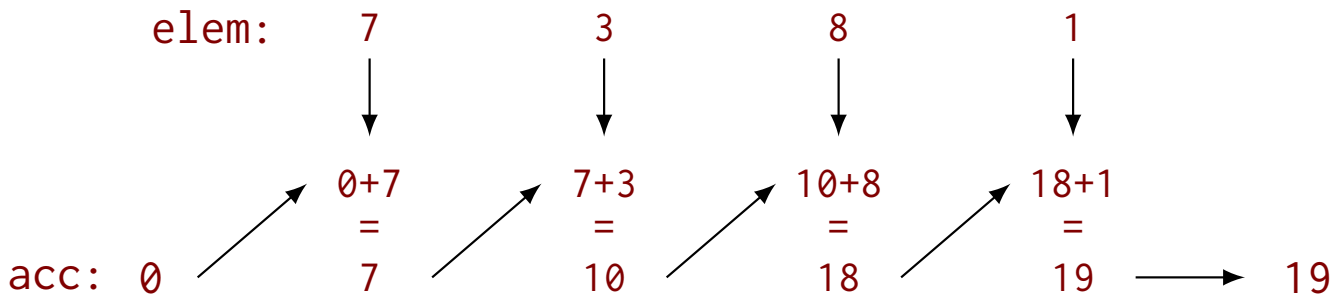
Ordre supérieur : les pliages (2/3)

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` :

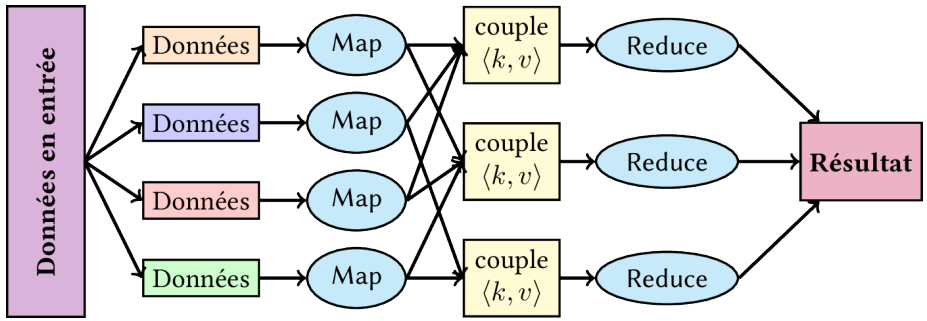
```
function treeSizeHigher(t) {
   const childrenSizes = map(treeSizeHigher, children(t));
   const sumSizes      = reduce((acc, el) ⇒ acc+el, childrenSizes, 0);
   return 1 + sumSizes;
}
```

- 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.



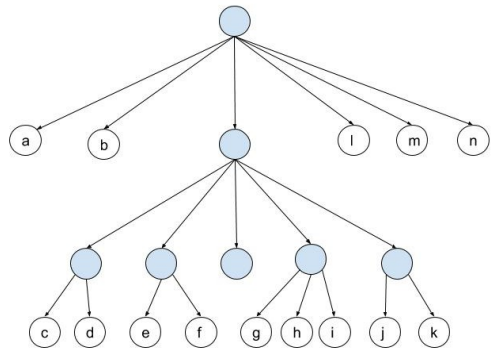
Source : Wikipedia

- 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.

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 . . .

♪ Vérifier du Ecmascript (1/2)

L'Ecmascript est un langage particulièrement libéral :

- aucune vérification des types ou des dépassements d'indice ;
- aucune vérification du nombre de paramètres pris par les fonctions ;
- insertion automatique de point-virgules si ils sont manquants.

D'où proviennent les problèmes dans les codes suivants ?

```
function cons(aCar, aCdr)          { return { car: aCar, cdr: aCdr }; }
function node(aVal, someChildren) { return { val: aVal, children: someChildren }; }

const aTree = node(7, cons(node(3, nil), cons(node(5, nil))));

const numC  = listLength(children(aTree)); // TypeError
```

♪ Vérifier du Ecmascript (1/2)

L'Ecmascript est un langage particulièrement libéral :

- aucune vérification des types ou des dépassements d'indice ;
- aucune vérification du nombre de paramètres pris par les fonctions ;
- insertion automatique de point-virgules si ils sont manquants.

D'où proviennent les problèmes dans les codes suivants ?

```
function cons(aCar, aCdr)          { return { car: aCar, cdr: aCdr }; }
function node(aVal, someChildren) { return { val: aVal, children: someChildren }; }

const aTree = node(7, cons(node(3, nil), cons(node(5, nil))));

const numC  = listLength(children(aTree)); // TypeError
```

Ici, le nombre de paramètres passé à `cons` est incorrect.

♪ Vérifier du Ecmascript (1/2)

L'Ecmascript est un langage particulièrement libéral :

- aucune vérification des types ou des dépassements d'indice ;
- aucune vérification du nombre de paramètres pris par les fonctions ;
- insertion automatique de point-virgules si ils sont manquants.

D'où proviennent les problèmes dans les codes suivants ?

```
function bizarreReturn() {
    return
           { car: 1 };
}

const strangeHead = bizarreReturn().car; // TypeError
```

♪ Vérifier du Ecmascript (1/2)

L'Ecmascript est un langage particulièrement libéral :

- aucune vérification des types ou des dépassements d'indice ;
- aucune vérification du nombre de paramètres pris par les fonctions ;
- insertion automatique de point-virgules si ils sont manquants.

D'où proviennent les problèmes dans les codes suivants ?

```
function bizarreReturn() {
    return ;
           { car: 1 };
}

const strangeHead = bizarreReturn().car; // TypeError
```

Ici, la machine virtuelle insère automatiquement un point-virgule.

Vérifier du Ecmascript (2/2)

Afin d'alléger les difficultés pour écrire du code correct, il existe des outils de **vérification**. En voici deux catégories :

- les vérificateurs **syntaxiques**, comme les linters ;
- les vérificateurs de **types**, usuellement dans les compilateurs.

Exemple : ESLint

ESLint est un outil de vérification de code Ecmascript.
Par exemple, sur le code `bizarreReturn` précédent :

```
3:12  error  Unreachable code                  no-unreachable
3:22  error  Unnecessary semicolon             no-extra-semi
```

La règle `no-unreachable` vérifie s'il n'existe pas de code après un `return`.

Vérifier du Ecmascript (2/2)

Afin d'alléger les difficultés pour écrire du code correct, il existe des outils de **vérification**. En voici deux catégories :

- les vérificateurs **syntaxiques**, comme les linters ;
- les vérificateurs de **types**, usuellement dans les compilateurs.

Exemple : le compilateur Typescript

Le compilateur `tsc` réalise une analyse des types.
Même sans aucune indication de type, il est capable de vérifier que les arguments passés à une fonction sont en nombre correct.

```
lists.ts - error TS2554: Expected 2 arguments, but got 1.
240 const aTree2 = node(7, cons(node(3, nil), cons(node(5, nil))));
                                                  ----------------
```

Citoyenneté de 1ère classe

Principe général

Écrire du code profitant du caractère hautement composable des fonctions.

- Les expressions mathématiques sont une premier signe de l'expressivité que l'on peut obtenir en composant des fonctions.
- Naturellement, on attend plus d'un langage de programmation.
- D'où la question de lister les qualités que l'on peut attendre des fonctions au sein des programmes $\Rightarrow$ **citoyenneté de 1ère classe**.

Définition (1ère classe)

Une construction d'un langage de programmation est dite **citoyenne de 1ère classe** (*1st class citizen*) si le langage l'autorise à :

- [**Nommage**] être nommée et affectée dans une variable ;
- [**Création**] être créée à la demande dans n'importe quel contexte ;
- [**Argument**] être passée comme argument à une fonction ;
- [**Retour**] être retournée comme résultat d'une fonction ;
- [**Stockage**] apparaître dans n'importe quelle structure de données.

- La 1ère classe n'est pas une fin en soi : certaines techniques sont appliquables même si toutes les propriétés ne sont pas vérifiées.
- Chaque propriété peut être vue comme une direction à explorer.
- La notion de 1ère classe peut s'appliquer à d'autres constructions que les fonctions : les macros, les modules, les classes ...

1ère ou 2ème classe ?

Exemple

En Ecmascript, les chaînes de caractères sont de 1ère classe :

```
let aString = "hello";                  // can be named, created and assigned
function repeat(str) {                  // can be passed to a function
  return str + str;  }                  // and returned from it
repeat(aString);                        //
let anArr = [ aString; "world" ];       // can be stored in a data structure
```

Contre-exemples

- En C, les fonctions **ne sont pas** de 1ère classe : on ne peut pas les créer à la demande.
- En C++, les objets sont de 1ère classe, mais les classes elle-mêmes **ne le sont pas**.

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

- **Retour** : être retournée comme résultat d'une fonction ;

```
function addOne(f) { return ((x) ⇒ f(x+1)); }
```

Les fonctions de 1ère classe

En Ecmascript, les fonctions **sont** des valeurs de 1ère classe :

- **Nommage** : être nommée et affectée dans une variable ;

```
let aFun = function (x) { return x + 1; };
```

- **Création** : être créée à la demande dans n'importe quel contexte ;

```
(function (x) { return x + 1; }) + 'is fun'
```

- **Argument** : être passée comme argument à une fonction ;

```
console.log(function (x) { return x + 1; });
```

- **Retour** : être retournée comme résultat d'une fonction ;

```
function addOne(f) { return ((x) ⇒ f(x+1)); }
```

- **Stockage** : apparaître dans n'importe quelle structure de données.

```
let anArray = [ Math.sin, (x) ⇒ x + 1 ];
```

Dans la suite, nous explorons chacune des propriétés de la 1ère classe :

- **Argument** : prendre en paramètre des fonctions ;
- **Retour** : être retourné comme résultat d'une fonction ;
- **Stockage** : apparaître dans une structure de données ;
- **Composition** : mélanger les précédentes propriétés.

1ère classe – Prendre en paramètre des fonctions

Dans quel but ?

- Une fonction est à considérer comme un morceau de code portable.
- Prendre une fonction en paramètre est donc largement plus flexible que prendre un entier ou une chaîne de caractères.

Définition (Fonction d'ordre supérieur)

Une **fonction d'ordre supérieur** (*higher-order function*) est une fonction dont au moins l'un des paramètres est une fonction.

Nous avons déjà rencontré de telles fonctions avec `map`, `filter` et `reduce`.
Quels besoins font apparaître naturellement ces fonctions dans le code ?

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

```
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;
}
```

Comment permettre de paramétrer le moment où la boucle s'arrête ?

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

```
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;
}
```

Comment permettre de paramétrer le moment où la boucle s'arrête ?

⇒ En ajoutant un paramètre fonctionnel `until` renvoyant un booléen.

Exemple de fonction d'ordre supérieur

La fonction `loop` est un exemple de fonction d'ordre supérieur qui répète un certain nombre de fois le `block` passé en paramètre :

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

La fonction `loop` peut s'écrire à partir de `loopUntil` comme suit :

```
let loop = (block, init, times) ⇒ loopUntil(block, init, (i) ⇒ i < times);
```

Définition (Généralisation)

Une **généralisation** fonctionnelle (*func. generalization*) d'une fonction f consiste à remplacer une partie du corps de f par un appel à une fonction qui est ajoutée aux paramètres de f.

- Les généralisations rendent le code plus adaptable et réutilisable.
- Elles se composent bien avec les fonctions anonymes qui peuvent fournir les paramètres fonctionnels supplémentaires.

Exemple de généralisation : countChars

Reprenons l'exemple de la fonction comptant les occurrences d'un caractère dans une chaîne :

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

Comment peut-on faire pour que `countChars` compte les caractères dans un ensemble donné (par exemple toutes les lettres de l'alphabet) ?

Exemple de généralisation : countChars

Reprenons l'exemple de la fonction comptant les occurrences d'un caractère dans une chaîne :

```
function countMatches(str, matchFun) { // matchFun is a 1-parameter function
    let res = 0;
    for (let i = 0; i < str.length; i++) {
        if (matchFun(str[i]))
          res += 1;
    }
    return res;
}
```

La fonction `countChars` peut s'écrire à partir de `countMatches` comme suit :

```
let countChars = (str, c) ⇒ countMatches(str, (s) ⇒ s === c);
```

Exemple de généralisation : sortThree

Prenons l'exemple d'une fonction triant un tableau de 3 éléments :

```
function sortThree([a, b, c]) {
    if (a < b) {  // a < b
        return (c < a) ? [c, a, b] :
               (c < b) ? [a, c, b] :
                         [a, b, c];
    } else {      // b <= a
        return (c < b) ? [c, b, a] :
               (c < a) ? [b, c, a] :
                         [b, a, c];
}}
```

Comment peut-on généraliser cette fonction ?

Exemple de généralisation : sortThree

Prenons l'exemple d'une fonction triant un tableau de 3 éléments :

```
function sortThreeGen([a, b, c], cmpFun) { // cmpFun is a 2-param function
    if (cmpFun(a, b)) { // a < b
        return (cmpFun(c, a)) ? [c, a, b] :
               (cmpFun(c, b)) ? [a, c, b] :
                                [a, b, c];
    } else {              // b <= a
        return (cmpFun(c, b)) ? [c, b, a] :
               (cmpFun(c, a)) ? [b, c, a] :
                                [b, a, c];
}}
```

La fonction `sortThree` peut s'écrire à partir de `sortThreeGen` comme suit :

```
let sortThree = (arr) ⇒ sortThreeGen(arr, (a, b) ⇒ a < b);
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : factorisation

Les généralisations peuvent servir à factoriser du code, par exemple :

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    const res = setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    const res = !setIsEmpty(aSet);
    numTests++;
    if (res) numPassed++;
    return res;
}
```

```
let numTests = 0;
let numPassed = 0;

function testEmptyIsEmpty() {
    const aSet = setEmpty();
    return setIsEmpty(aSet);
}
function testNonEmptyIsNotEmpty() {
    const aSet = setEmpty();
    setAdd(aSet, 1);
    return !setIsEmpty(aSet);
}
function runTest(aTest) {
    const res = aTest();
    numTests++;
    if (res) numPassed++;
    return res;
}
```

Exemple de généralisation : stratégie

Les généralisations peuvent servir à définir des **stratégies** :

```
function playHeadsOrTails() {
    let gain = 10;
    while ((gain > 0) && (gain < 20)) {
        let bet = gain;
        gain += (Math.random() > 0.5) ? bet : -bet;
    }
    return gain;
}
```

Comment peut-on généraliser cette fonction ?

Exemple de généralisation : stratégie

Les généralisations peuvent servir à définir des **stratégies** :

```
function playHeadsOrTailsGen(betFun, contFun) { // betFun and contFun
    let gain = 10;                               // are 1-parameter functions
    while (contFun(gain)) {
        let bet = betFun(gain);
        gain += (Math.random() > 0.5) ? bet : -bet;
    }
    return gain;
}
```

La fonction `playHeadsOrTails` peut s'écrire à partir de l'autre comme suit :

```
let playHeadsOrTails = () => playHeadsOrTailsGen((a) => a , (a) => a <20);
```

Généralisations et généricité

Définition (Généricité – informelle)

Une fonction est dite **générique** si elle peut être utilisée correctement sur différents *types* de valeurs.

Exemple

La dernière fonction `countMatches` est en fait une fonction générique :

```
countMatches("abcde",      (c) ⇒ c>="a" && c<="e"); // → 5
countMatches([1,2,3,4,5], (c) ⇒ c%2 === 0);        // → 2
```

- Les généralisations entraînent de la généricité.
- Cette généricité est une forme particulière de **polymorphisme**.

```
function countMatches<T>(arr: T[], match: (T) ⇒ boolean) : number
```

Caveat sur les généralisations :

- Il y a un compromis entre ajouter des paramètres fonctionnels et garder des prototypes de fonctions raisonnables.
- Nous verrons plus tard d'autres techniques pour transporter plusieurs paramètres fonctionnels en même temps.

1ère classe – Renvoyer des fonctions

Dans quel but ?

- Une fonction renvoyant une fonction, c'est un **constructeur** de fonctions.
- Si les fonctions sont considérées comme des valeurs centrales, avoir des facilités pour les construire est un bonus.
- Mieux, donner des paramètres à ces constructeurs permet d'adapter les fonctions ainsi construites.

Nous avons déjà rencontré de tels exemples dans les slides précédents.
Tâchons de mettre des noms sur les techniques associées.

Spécialisation (1/2)

Définition (Spécialisation)

Une **spécialisation** fonctionnelle (*specialization*) d'une fonction f sur un de ses paramètres p consiste à retirer p des paramètres de f et le remplacer dans le corps de f par une valeur donnée.

Exemple

```
function add(x, y) { return x + y; }
function add0ne(y) { return add(1, y);} // Specialization of add on x = 1
```

- Il est aussi possible d'utiliser le terme d'**application partielle**.

Spécialisation (2/2)

- Une spécialisation peut être réalisée à l'aide de **paramètres par défaut** :

```
function add(x, y = 1) { return x + y; }
add(5);          // → 6
```

- Une spécialisation peut être réalisée à l'aide de **fonctions anonymes** :

```
const addOne = function (x) { return add(x, 1); }
addOne(5);       // → 6
```

(e.g pour les langages ne permettant pas les paramètres par défaut)

- Mieux, il est possible de **paramétrer** la spécialisation :

```
const addAny = (v) ⇒ function (x) { return add(x, v); }
addAny(660)(6);  // → 666,        addAny(660) is the function ''adding 660''
```

Curryfication (1/3)

Définition (Forme curryfiée)

La **forme curryfiée** d'une fonction (*curried form*) est une fonction réalisant le même calcul, prenant ses paramètres un par un dans des fonctions successives :

$(x_1,\ x_2,\ \ldots,\ x_n) \Rightarrow \{\ \text{body}\ \} \quad \rightarrow \quad (x_1) \Rightarrow (x_2) \Rightarrow \ldots \Rightarrow (x_n) \Rightarrow \{\ \text{body}\ \}$

Exemple

```
function add(x, y) { return x + y; }                    // uncurried form

function addCurrified1(x) {                             // curried form (function)
    return function (y) { return x + y; }               //
}                                                       //

const addCurrified2 = (x) => (y) => { return x + y; }; // curried form (anonymous)
```

Curryfication (2/3)

- Tant qu'elle n'a pas reçu tous ses paramètres, l'appel à une fonction curryfiée renvoie une fonction :

```
function log = (lvl) ⇒ (date) ⇒ (msg) ⇒ `${lvl} (${date}): ${msg}`;
log('DEBUG');                                  // → Function
log('DEBUG')('12:34');                         // → Function
log('DEBUG')('12:34')('The server is on fire'); // → String
```

Ce sont des exemples de fonctions renvoyant des fonctions.

- La curryfication est une forme de **contrôle de l'évaluation** : le calcul n'est effectué qu'une fois le dernier argument fourni.
- Les fonctions curryfiées sont facilement spécialisables :

```
const debug = log('DEBUG');              // Specialize on first parameter
debug('13:56')('Out of water error');    // Reuse the specialized function
```

Curryfication (3/3)

Il est possible de curryfier automatiquement une fonction en Ecmascript.
(mais cela ne fait pas partie de la bibliothèque standard)

Extrait de la documentation de la fonction `curry` de la bibliothèque `lodash` :

```
import _ from 'lodash';
```

```
const abc = function(a, b, c) { return [a, b, c]; };

const curried = _.curry(abc);   // transform into curried form

typeof curried(1); // "function"
curried(1)(2)(3);  // ⇒ [1, 2, 3]
curried(1, 2)(3);  // ⇒ [1, 2, 3]
curried(1, 2, 3);  // ⇒ [1, 2, 3]
```

Dans certains langages de programmation (OCaml, Haskell, ...), les fonctions sont automatiquement curryfiées.

Généralisations et spécialisations

Les généralisations et les spécialisations se complètent naturellement :

A generalization

```
function countMatches(coll, matchFun) { // matchFun is a 1-parameter function
    let res = 0;
    for (let i = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;
}
```

Some specializations

```
// Tells if a string contains dangerous HTML characters such as brackets
const hasBrackets  = (coll) ⇒ countMatches(coll, (c) ⇒ /[<>]/.test(c)) > 0;
// Tells if an array contains undefined elements
const hasUndefined = (coll) ⇒ countMatches(coll, (c) ⇒ c === undefined) > 0;
```

Exemple de générateurs : pluck

La fonction `pluck` permet de créer aisément des accesseurs dans des objets :

```
function pluck(aField) {                    // Example of a curried function
    return function(obj) {                  // taken from [Fogus]
        return (obj && obj[aField]);
    };}
```

Exemple de générateurs : pluck

La fonction `pluck` permet de créer aisément des accesseurs dans des objets :

```
function pluck(aField) {                    // Example of a curried function
    return function(obj) {                  // taken from [Fogus]
        return (obj && obj[aField]);
    };}
```

Quelques manipulations d'une bibliothèque de livres :

```
const books = [
    { id: 7, title: 'Ender\'s game', author: 'Scott Card Orson' },
    { id: 1, title: 'Wyrd sisters',  author: 'Pratchett Terry' },
    { id: 9, title: 'Elevation',     author: 'Brin David'  },          ];
```

```
const getTitle  = pluck('title');
books.map(getTitle); // → [ "Ender's game", 'Wyrd sisters', 'Elevation' ]
const getAuthor  = pluck('author');
books.filter((b) ⇒ getAuthor(b).startsWith('P'));
                    // → [ { title: 'Wyrd sisters', author: 'Pratchett Terry' } ]
```

Exemple de générateurs : compareWith

La fonction `compareWith` construit des comparateurs utilisables avec `sort` :

```
function compareWith(aField) {
    return function(obj1, obj2) {
        const v1 = pluck(aField)(obj1), v2 = pluck(aField)(obj2);
        return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
    };}
```

Exemple de générateurs : compareWith

La fonction `compareWith` construit des comparateurs utilisables avec `sort` :

```
function compareWith(aField) {
    return function(obj1, obj2) {
        const v1 = pluck(aField)(obj1), v2 = pluck(aField)(obj2);
        return (v1 < v2) ? -1 : (v1 > v2) ? 1 : 0;
    };}
```

Quelques exemples de tris avec la bibliothèque précédente :

```
function sortArray(arr, cmp) {                  // return a sorted copy of an array
    const arrC = [...arr];
    arrC.sort(cmp);
    return arrC;
}
sortArray(books, compareWith('author'));    // books sorted with ids [9, 1, 7]
sortArray(books, compareWith('title'));     // books sorted with ids [9, 7, 1]
```

Les comparateurs peuvent même être étendus de manières variées :

- Pour inverser un comparateur :

```
function compareWithReverse(field) {
    return function(obj1, obj2) {
        return - compareWith(field)(obj1, obj2);
    };}
sortArray(books, compareWithReverse('title'));
```

- Pour composer plusieurs comparateurs :

```
function compareWithFields(...fields) {
    return function(obj1, obj2) {
        return fields.reduce((b, aField) => {
            return (b !== 0) ? b : compareWith(aField)(obj1, obj2);
        }, 0);
    };}
sortArray(books, compareWithFields('id', 'title', 'author'));
```

Exemple : décorateurs

Définition (Décorateur)

Un **décorateur** fonctionnel prend une fonction f et renvoie une fonction avec le même prototype, avec un comportement augmenté.

Par exemple, pour logguer les appels à une fonction :

```
function log(aFun, aMsg) {
    return function(...args) {
        console.log(`${aMsg} : ${aFun.name}(${args})`);
        const res = aFun(...args);                    // Call aFun
        console.log(`${aMsg} returns ${res}`);
        return res;
}} // Beware when using with recursive functions
```

```
function fact(n) {return (n<=1) ? 1 : n*fact(n-1); }
fact = log(fact, "Fact");          // Replace 'fact'
fact(5);                  // Calls are properly logged
```

```
Fact : fact(3), Fact : fact(2)
Fact : fact(1), Fact returns 1
Fact returns 2, Fact returns 6
```

À comparer à la fonction `wrap` des bibliothèques Underscore et Lodash.

Conclusion (partielle) sur la 1ère classe

- La 1ère classe est une propriété listant un ensemble de modes d'utilisation des objets, ici les fonctions.
- A chacun de ces modes d'utilisations correspondent des techniques de programmation.
  - Les **généralisations** utilisent des paramètres fonctionnels pour écrire du code générique, réutilisable.
  - Les **spécialisations** renvoient des fonctions dans lesquelles on a fixé les valeurs de certains paramètres.
- Généralisations et spécialisations fonctionnent seules, mais se combinent avec profit, les secondes aidant à la réutilisation des premières.

Fonctions et arguments en Ecmascript

En Ecmascript, au sein d'une fonction, il est possible de connaître l'ensemble des arguments qui lui sont passés grâce à la variable `arguments` :

```
function myConcat(separator) {                    // Adapted from developer.mozilla.org
    console.log(arguments);                         // display arguments
    const strings  = Array.from(arguments).slice(1); // remove first arg
    return strings.join(separator);                 // join args with first arg
}
```

```
myConcat(',␣')                  // Log : [Arguments] { '0':', ' }
                                // → ''
myConcat(',␣','aa','bb','cc') // Log : [Arguments] { '0':', ', '1':'aa', '2':'bb', '3':'cc' }
                                // → 'aa, bb, cc'
```

Fonctions et arguments en Ecmascript

En Ecmascript, au sein d'une fonction, il est possible de connaître l'ensemble des arguments qui lui sont passés grâce à la variable `arguments` :

```
function myConcat(separator, ..strings) {       // Adapted from developer.mozilla.org
    console.log(arguments);                          // display arguments

    return strings.join(separator);                  // join args with first arg
}
```

```
myConcat(',␣')                  // Log : [Arguments] { '0':', ' }
                                // → ''
myConcat(',␣','aa','bb','cc')   // Log : [Arguments] { '0':', ', '1':'aa', '2':'bb', '3':'cc' }
                                // → 'aa, bb, cc'
```

Fonctions et arguments en Ecmascript

En Ecmascript, au sein d'une fonction, il est possible de connaître l'ensemble des arguments qui lui sont passés grâce à la variable `arguments` :

```
function myConcat(separator, ..strings) {       // Adapted from developer.mozilla.org
    console.log(arguments);                          // display arguments

    return strings.join(separator);                  // join args with first arg
}
```

```
myConcat(',␣')                  // Log : [Arguments] { '0':', ' }
                                // → ''
myConcat(',␣','aa','bb','cc')   // Log : [Arguments] { '0':', ', '1':'aa', '2':'bb', '3':'cc' }
                                // → 'aa, bb, cc'
```

- Les fonctions à nombre variable d'arguments sont fréquentes en Ecmascript.
- Utiliser la reconnaissance de motifs (`..args`) permet de rendre la spécification des arguments plus explicite.

1ère classe – Stocker des fonctions

Dans quel but ?

- Plusieurs fonctions réunies ensemble dans une même structure peuvent facilement se **composer** ou plus généralement **partager** des liens ;

  Comment tirer profit des liens ainsi créés ?

- Ranger des fonctions dans des structures de données, c'est considérer les fonctions comme des données **dynamiques** :

  transportables, partageables, réutilisables . . .

  Quels genre de données profitent de composants fonctionnels ?

Exemple : codage et décodage

Reprenons l'exemple de fonctions de chiffrement et déchiffrement :

```
function makeCypher(key) {
    function cypher (str, offset) { ... }
    function encode(str) { return cypher(str, key);  }
    function decode(str) { return cypher(str, -key); }
    return { encode: encode, decode: decode };
}

const c = makeCypher(42);
c.decode(c.encode('Hello')); // → 'Hello'
```

- Les deux fonctions `encode` et `decode` sont inverses l'une de l'autre.
- Il est sensé de les garder ensemble et proches dans un dictionnaire.

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {




    return { re: re,       im: im };

}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1;
// → { re: 1, im: 0 }

cI;
// → { re: 0, im: 1 }
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {




    return { re: re,       im: im };

}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → Error

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {



    }




    return { re: re,        im: im,
             add: add };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → undefined

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {
        return makeComplex(c.re + re,
                           c.im + im);
    }

    return { re: re,       im: im,
             add: add };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → { re: 1, im: 1, add.. }

cI.mul(cI);
// → Error
```

Exemple : objets-enregistrements (1/2)

Les **enregistrements** (`record`, `struct` ...) permettent aisément de regrouper ensemble des données et des traitements (ici des nombres complexes) :

```
function makeComplex(re, im) {
    function add(c) {
        return makeComplex(c.re + re,
                           c.im + im);
    }
    function mul(c) {
        return makeComplex(c.re*re - c.im*im,
                           c.re*im + c.im*re);
    }
    return { re: re,       im: im,
             add: add,     mul: mul };
}
```

```
let c1 = makeComplex(1,0);
let cI = makeComplex(0,1);

c1.add(cI);
// → { re: 1, im: 1, add, mul.. }

cI.mul(cI);
// → { re: -1, im: 0, add, mul.. }
```

Exemple : objets-enregistrements (2/2)

Les **enregistrements** permettent aisément de regrouper ensemble des données et des traitements (ici des données génériques dans un ensemble) :

```
struct generic_t {
   int (*cmp)(const void *, const void *); // a function for comparing values
   void* (*copy)(const void *);            // a function to copy values
   void (*del)(void *);                    // a function to free values
};

struct set* set__empty(const struct generic_t*);   // allocates a generic set
```

- Le résultat est une structure de données contenant des fonctions.
- Il est ainsi possible de créer des valeurs ressemblant à des **objets** (sans la complexité du monde objet et l'accès au `this`).

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Exemple : modules

Définition (Module)

Un **module** est un espace de nommage avec des possibilités de masquer ou non les éléments à l'intérieur.

Le pattern 'Module' en Ecmascript

```
const aModule = ( function () {
    const conf = { caching : true,
                   language : 'en' };
    return {
        reportConfig :  function () {
            console.log(`Caching ${conf.caching}`); },
        updateCaching : function (caching) {
            conf.caching = caching; }
    }
} ) ();
aModule.updateCaching(false);
aModule.reportConfig(); // Log : 'Caching false'
```

Private

Public

Caveat sur le stockage de fonctions

Les constructions ainsi évoquées sont souvent des **simplifications**.
Il existe des constructions plus complexes à un tel niveau de granularité :

- la couche **objet** d'Ecmascript est plus riche que ce que peuvent exprimer les enregistrements (prototypes, accès au `this` ...)
- la couche **modulaire** d'Ecmascript a évolué et apporte plus de fonctionnalités (gestion des dépendances, chargement asynchrone ...)

Néanmoins, elles permettent de faire le lien entre les différents styles de programmation.

Et elles démontrent l'intérêt de stocker ensemble des fonctions.

Callbacks (1/2)

Les fonctions peuvent aussi être utilisées par et pour elle-mêmes.

Définition (Callback)

Un **callback** est une fonction destinée à être exécutée lorsqu'un événement particulier se produit au cours de l'exécution.

- Les navigateurs Web représentent les pages HTML sous la forme d'un arbre suivant une spécification nommée **Document Object Model** ;
- Cette spécification définit en particulier un ensemble d'événements auxquels attacher des callbacks :

  `click`, `drag`, `drop`, `focus`, `input`, `keypress`, `load` ...

- Le langage permet alors d'associer événements et callbacks :

```
const button = document.querySelector('button');
button.addEventListener('click',
    event ⇒ { button.innerHTML = `#Clicks : ${event.detail}`; });
```

Callbacks (2/2)

Les utilisations des callbacks ne se limitent pas aux interfaces graphiques.

- De nombreuses fonctions de l'API de Node.js utilisent des callbacks pour gérer les erreurs (*error callback convention*).
- En particulier, les opérations fonctionnant de manière asynchrone, comme celles faisant appel au système de fichiers :

```
const aPath = '/some/doomed/file';          // fs.unlink deletes a file
fs.unlink(aPath, (err) ⇒ {                  // callback called on completion
  if (err) throw err;                       // re-throw the error if ever
  console.log(`successfully deleted '${aPath}'`); // otherwise just log
});
```

Un paramètre fonctionnel ajouté à une fonction et censé être appelé à sa terminaison s'appelle une **continuation**.

Deux mots sur les continuations

Pour simplifier, une **continuation** est un callback appelé après le retour de la fonction sur son résultat. Par exemple :

Version classique

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



[3,4,5,6].map(fibo); // → [3,5,8,13]
```

Version avec continuations

```
function fiboCps(n, cont) {
  if (n <= 1) {
    return cont(1);
  } else {
    return fiboCps(n-2, (a) ⇒
           fiboCps(n-1, (b) ⇒
                        cont(a+b)
           ));
} }

const myFibo = (e) ⇒ fiboCps(e, (r) ⇒ r);
[3,4,5,6].map(myFibo); // → [3,5,8,13]
```

Ce style d'écriture de code est appelé **continuation-passing style** ou CPS.

Représentation des données par les fonctions (1/3)

Les fonctions peuvent servir pour représenter des données par elle-mêmes.

- prédicats,
- comparateurs,
- stratégies,
- callbacks,
- ...

En tant que données, les fonctions brillent surtout par leur **expressivité**.

Représentation des données par les fonctions (2/3)

Idée

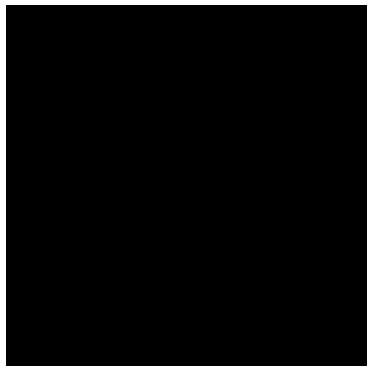
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

```
function setFull(p) {
  return true;
}
```



Représentation des données par les fonctions (2/3)

Idée

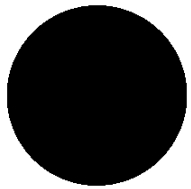
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

```
function setCircle(p) {
  return ((p.x-1)**2 + (p.y-1)**2) <= 0.25;
}
```



Représentation des données par les fonctions (2/3)

Idée

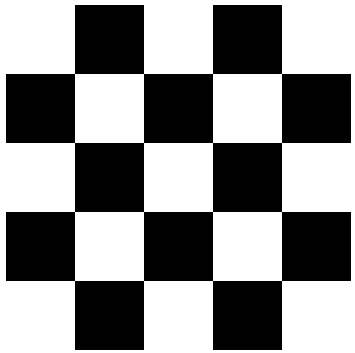
User de **fonctions caractéristiques** pour décrire des ensembles complexes.

Étant donné un ensemble E, sa fonction caractéristique est la fonction χ_E :

$$\chi_E(t) = \begin{cases} 1 & \text{si } t \in E \\ 0 & \text{sinon} \end{cases}$$

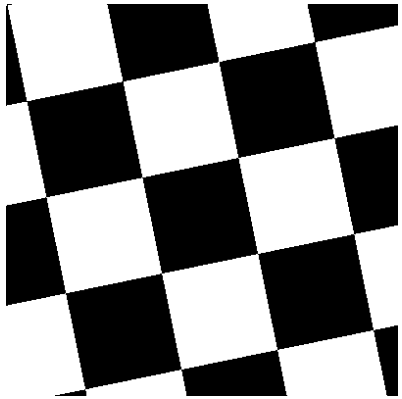
- Ces fonctions permettent de représenter des ensembles de **données** de manière générique.
- Le type des données et la forme des ensembles sont très versatiles :

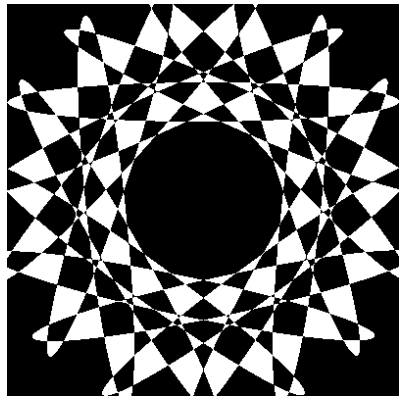
```
function setGrid(p) {
  let rx = Math.floor(x*2.5);
  let ry = Math.floor(y*2.5);
  return (rx+ry) % 2 == 1;
}
```



Représentation des données par les fonctions (3/3)

Ces fonctions peuvent ensuite être transformées en d'autres fonctions.
Quelques exemples pour les ensembles de points :





Idée

Représenter les données par des fonctions offrir des possibilités étendues par rapport aux valeurs classiques.

- Générateur de listes de lectures sur critères de préférence ;
- Familles de stratégies (MinMax, ANN ...) pour les jeux ...

Exemple : validateurs [Fogus]

Idée

Créer des **validateurs** vérifiant si les valeurs ont des propriétés adéquates.

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

```
function validator(message, fun) {
  const f = function(...args) { return fun(...args); };
  f['message'] = message;
  return f;
}

function checker(...validators) {
  return function(obj) {
    return _.reduce(validators,
                    function(errs, aValidator) {
      return (aValidator(obj)) ?
                errs : [ ...errs, aValidator.message ];
    }, []);
  };
}
```

```
const argNumber = validator(
    "arg should be a number", _.isNumber);
const argPos = validator(
    "arg should be > 0", (n) => n > 0);
const argEven = validator(
    "arg should be even", (n) => n % 2 === 0);

// A checker that combines 2 validators
const chk1 = checker(argNumber, argPos);

// Examples
chk1(5);     // → []
chk1(-3);    // → ['arg should be > 0']
chk1("lol"); // → ['arg should be a number',
             //    'arg should be > 0']
```

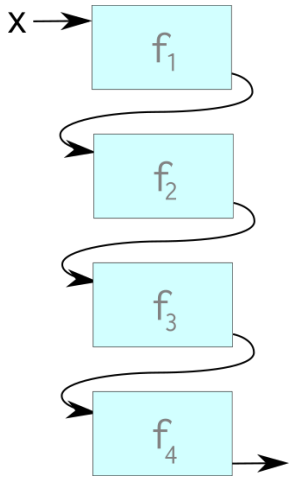
Profite ici de la facilité de composer des fonctions simples (les validateurs) en des fonctions complexes (le checker).

Composer des fonctions

Dans quel but ?

- Composer des fonctions entre elles peut sembler une évidence, mais on a souvent tendance à penser les fonctions de manière indépendante.

  Quelles techniques permettent de profiter de cette capacité de composition ?



- Accessoirement, comment concevoir des bibliothèques de fonctions qui facilitent ces compositions ?

Exemple : pipeline

Composer les fonctions de manière à ce que chacune envoie son résultat en paramètre à la suivante peut se faire facilement avec un **pipeline** :

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

```
function pipeline(seed, ...args) {
    return _.reduce(args,
        function(l,r) { return r(l); },
    seed);
};

pipeline("abba est un groupe suedois",
         (s) ⇒ s.split(" "),          // → ["abba", "est", "un", ... ]
         (a) ⇒ a.map(capitalize),     // → ["Abba", "Est", "Un", ... ]
         (a) ⇒ a.join(' '));          // → "Abba Est Un Groupe Suedois"
```

Principe d'un pipeline

Une donnée qui est transformée au fur et à mesure d'un calcul.

Exemple : chain avec lodash

Plusieurs bibliothèques proposent de composer les fonctions en pipeline.
Extrait de la documentation de la fonction `chain` de la bibliothèque `lodash` :

```
import * as _ from 'lodash-es';
```

```
const users = [ { 'user': 'barney',  'age': 36 },
                { 'user': 'fred',    'age': 40 },
                { 'user': 'pebbles', 'age': 1  } ];

const youngest = _
  .chain(users)
  .sortBy('age')
  .map((o) => `${o.user} is ${o.age}`)
  .head()
  .value();                                  // => 'pebbles is 1'
```

- Le chaînage est effectué ici en chaînant les appels de méthodes.
- La fonction `chain` débute la chaîne et la méthode `value` la termine.

Exemple : ifElse avec Ramda

La composition peut même se permettre de ne pas être linéaire.
Exemple d'utilisation de la fonction `ifElse` de la bibliothèque `Ramda` :

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

```
const addOrRemove = val ⇒                // Example adapted from R.W. Pearce
    _.ifElse(
        _.includes(val),                 // (arr) ⇒ is val in arr ?
        _.without(_.of(val)),            // (arr) ⇒ arr without [val]
        _.append(val)                    // (arr) ⇒ arr with val appended
);

addOrRemove(7)([1,2,3,4]); // → [ 1, 2, 3, 4, 7 ]
addOrRemove(3)([1,2,3,4]); // → [ 1, 2, 4 ]
```

- Avec `Ramda`, toutes les fonctions sont curryfiées par défaut.
- La curryfication simplifie la composition des fonctions.

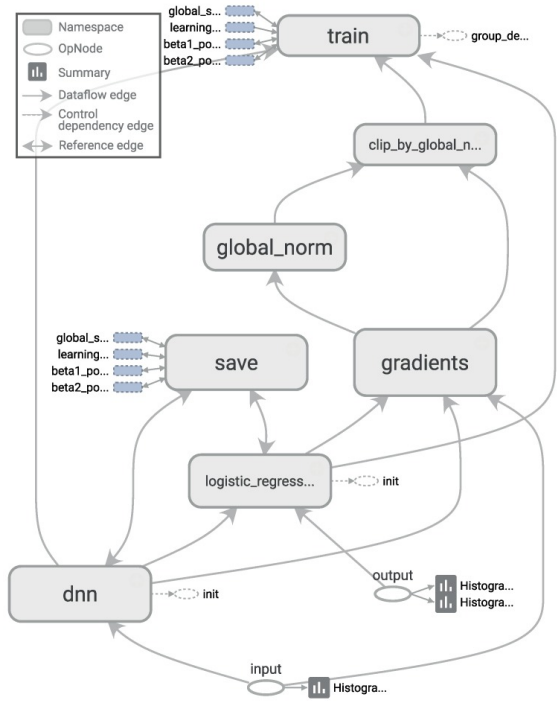
Data Flow Programming

- Le style de programmation ainsi obtenu est orienté données (*data flow*) plutôt que contrôle (*control flow*).

- Il existe un paradigme de programmation orienté autour des flots de données (*dataflow programming*), structurés par un graphe.

- En particulier utilisé dans la bibliothèque `tensorflow` en apprentissage automatique.

- À contraster avec le style impératif classique.



Source : Y. Saez et. al.
DOI : 10.3390/s17010066

Map-Reduce

Les calculs sur les tableaux se chaînent facilement :

```
const data = [ { desc: 'Chocolate',  amount: 3, unitPrice: 10,  type: 'expense' },
               { desc: 'Tournament', amount: 1, unitPrice: 50,  type: 'asset' },
               { desc: 'Candy',      amount: 7, unitPrice: 3,   type: 'expense' },
               { desc: 'Lottery',    amount: 1, unitPrice: 100, type: 'asset' } ];
```

Version impérative

```
let total = 0;
for (let i = 0; i < data.length; i++) {
  if (data[i].type === 'expense'){
    let expense = data[i].amount *
                  data[i].unitPrice;
    total += expense;
  }
}
```

Version fonctionnelle / composition

```
const total = data
  .filter((line) => {
    return line.type === 'expense';
}).map((line) => {
    return line.amount *
           line.unitPrice;
}).reduce((sum, expense) => {
    return sum + expense; });
```

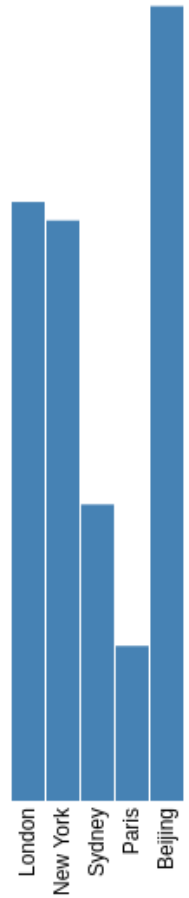
- Peut perdre en performance si chaque objet intermédiaire est construit.
- Pour mitiger cela, les calculs peuvent s'effectuer de manière **paresseuse**.

Fluent interfaces

Le chaînage peut devenir une propriété de l'interface de programmation, comme dans le cas des **fluent interfaces**.

L'exemple suivant est réalisé en `D3.js` :

```
const cities = [
        { name: 'London',   population: 8674000},
        { name: 'New York', population: 8406000},
        // ...
                ];
d3.selectAll('rect')
  .data(cities)
  .attr('height', 19)
  .attr('width', function(d) {
    var scaleFactor = 0.00004;
    return d.population * scaleFactor;
  })
  .attr('y', function(d, i) {
    return i * 20;
  });
```



Source : https://www.d3indepth.com/datajoins

Promises API

Les **promesses** (*promises*) permettent de chaîner des calculs asynchrones :

```
getJSON('https://web.site/story.json').     // Example from https://web.dev/promises
    then(function(story) {
      return getJSON(story.chapterUrls[0]);
    }).
    then(function(chapter1) {
      addHtmlToPage(chapter1);
    }).
    catch(function(error) {
      addTextToPage(`Failed to show chapter : ${error}`);
    }).
    then(function() {
      document.querySelector('.spinner').style.display = 'none';
    });
```

- Il faut voir ici les promesses comme une manière de **préparer un calcul**. Comme lorsqu'on compose des fonctions ou séquence des instructions.

L'inévitable slide sur les monades

La composition des fonctions peut être facilité par des choix de syntaxe.
L'exemple suivant en Haskell est un code fonctionnel pur :

```
battle = do                                   -- Adapted from www.haskellforall.com
    forM_ ["Take␣that!", "and␣that!", "and␣that!"] $ \taunt → do -- Charge!
        lift $ putStrLn taunt
        boss.health -= 10

    fireBreath (Point 0.5 1.5)                -- The dragon awakes !

    lift $ putStrLn "Retreat!"                -- Retreat is the better part of valor
    zoom (units.traversed.position) $ do
        x += 10                               -- Warning : this code is not 'vanilla' Haskell
        y += 10                               -- It makes usage of monads and lenses
```

Exemple tiré du blog Haskell For All

- Les calculs effectués ici sont des compositions de fonctions pures.
- Elles se transmettent un état initial et renvoient un état final.

Conclusion sur la 1ère classe

En suivant les idées de la 1ère classe, nous avons étudié des manières particulières d'utiliser les fonctions dans des structures :

- Les fonctions peuvent **représenter des valeurs complexes** en tant que tels (ensembles, prédicats, callbacks . . .) ;
- Si on les rassemble dans des structures, elles forment un début d'**architecture** (modules, objets . . .) ;

Un autre aspect remarquable tient dans la facilité à composer les fonctions.

- Les algorithmes peuvent être pensés comme des **compositions** de calculs, plutôt que des suites d'instructions (e.g data-flow).
- Les possibilités en terme de syntaxe et de forme d'API peuvent faciliter voire encourager un style fonctionnel.

Techniques de programmation

Dans ce chapitre, nous allons présenter des techniques “avancées” liées à la programmation fonctionnelle :

- la programmation typée avec des fonctions ;
- la programmation dirigée par les fonctions ;
- les techniques de contrôle de l'évaluation.

Ces techniques ne font pas partie des fondamentaux de la programmation fonctionnelle, mais illustrent des manières supplémentaires de l'utiliser.

Programmation typée

Quelles interactions entre prog. typée et prog. fonctionnelle ?

Principes de la programmation typée

- Classer les expressions apparaissant dans un programme selon des **types**,
- Vérifier que le composition de ces **types** au sein du programme vérifie un ensemble de règles de cohérence.

Contre-exemple illustratif

```
locomotive + flower
```

- Ecmascript est un langage à typage **dynamique** : les types existent, mais ne sont vérifiés qu'à l'exécution.

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s) {                                    // Javascript version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
TypeError: s.split is not a function                     // Execution with node
```

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s : any) : any {                    // Typescript 'any' version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
TypeError: s.split is not a function                 // Execution with node
```

Typescript

- Typescript est un langage développé par Microsoft et se présentant comme un sur-ensemble d'Ecmascript ajoutant une couche de **typage**.
- Le compilateur `tsc` transforme un fichier `.ts` en `.js`, tout en réalisant un ensemble de vérifications **statiques** de ces règles de cohérence.
- Les indications de types y sont optionnelles et peuvent être ajoutées de manière **graduelle**.

```
function reverse(s : string) : string {          // Typescript 'typed' version
  return s.split('').reverse().join('');
}
reverse('hello world'); // → 'dlrow olleh'
reverse(456);           // → ???
```

```
error TS: type 'number' not assignable to type 'string'.    // Compilation
```

Les types en Typescript

Pour référence, les types des valeurs de base en Typescript sont :

- les valeurs non initialisées
  et les valeurs nulles, vides ou inexistantes ; `undefined` `null`
- les valeurs booléennes ; `boolean`
- les chaînes de caractères immutables `string`
  et les types des clés des dictionnaires ; `symbol`
- les nombres flottants `number`
  et les grands nombres ; `bigint`
- et les **objets** :
  - les dictionnaires clé-valeurs ; `{ age: number; name: string }`
  - les tableaux ; `number[]` / `string[]`
  - les fonctions. `(param: string) ⇒ number`

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches   (coll,
                         matchFun                    ) {
    let res = 0;
    for (let i = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches<T>(coll : ArrayLike<T>,
                         matchFun : (T) ⇒ boolean) : number {
    let res : number = 0;
    for (let i : number = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

Fonctions et polymorphisme

- Programmer avec des fonctions, en particulier des paramètres fonctionnels a tendance à produire du code **générique**.
- En terme de types, cette généricité est une forme de **polymorphisme** : le code peut s'appliquer à des valeurs de types différents.
- Concrètement, il s'agit de polymorphisme paramétrique, se traduisant par la présence de **variables de types**.

```
function countMatches<T>(coll : ArrayLike<T>,
                         matchFun : (T) ⇒ boolean) : number {
    let res : number = 0;
    for (let i : number = 0; i < coll.length; i++) {
        if (matchFun(coll[i]))
          res += 1;
    }
    return res;                                          }
```

`T` est une variable de type, et peut être
- un caractère (si `coll : string`)
- un nombre (si `coll : number[]`)

Faire de la programmation fonctionnelle dans un cadre typé implique forcément de rencontrer des types paramétriques :

- En Haskell : Lien

```haskell
map :: (a → b) → [a] → [b]
zip :: [a] → [b] → [(a, b)]
```

- En Java (ici sur les `Stream`) : Lien

```java
public interface Stream<T>
void        forEach(Consumer<? super T> action)
Stream<R>   map(Function<? super T, ? extends R> mapper)
```

- Voire même en Ecmascript, par ex. avec la bibliothèque `Ramda` : Lien

```haskell
map :: Functor f ⇒ (a → b) → f a → f b
zip :: [a] → [b] → [[a,b]]
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Il est possible de traiter les valeurs de type `Shape` au cas par cas :

```
function getArea(shape: Shape) {
    switch (shape.kind) {
        case "circle":     return Math.PI * shape.radius ** 2;
        case "square":     return shape.sideLength ** 2;
        case "rectangle":  return shape.h * shape.w;
  } }
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Il est aussi possible de vérifier ici si les fonctions sont **exhaustives** :

```
function getArea(shape: Shape) {
    switch (shape.kind) {
        case "circle":     return Math.PI * shape.radius ** 2;
        case "square":     return shape.sideLength ** 2;
        default:           const unused : never = shape;  // Never reached ?
  } }
```

Exemple de typage : types sommes

Dans cet exemple en Typescript, le type `Shape` est un **type somme**.
Les valeurs de ce type peuvent prendre 4 formes distinctes.

```
interface Circle    {  kind: "circle";     radius: number;  };
interface Square    {  kind: "square";     sideLength: number;  };
interface Rectangle {  kind: "rectangle";  h: number;  w: number; };
interface Triangle  {  kind: "triangle";   sides: number[]; };

// Shape is either one of Circle, Square, Rectangle or Triangle (discriminated union)
type Shape = Circle | Square | Rectangle | Triangle;
```

Le compilateur `tsc` vérifie alors et propose les cas manquants :

```
error TS: Type 'Rectangle␣|␣Triangle' is not assignable to type 'never'.
```

Avantages et inconvénients de la prog. typée

Typer ses programmes aide à rendre leur comportement explicite.

- Des outils de **vérification** (analyse statique) peuvent être utilisés pour assurer des propriétés de **sûreté** sans exécuter le code.
- Les types servent aussi de **documentation** (minimale) pour les programmeurs, en le spécifiant de manière systématique et succinte.

Ils peuvent être aussi vus comme des freins au développement :

- La programmation sans les types est certes fragile, mais permet d'écrire du code très **rapidement** (e.g prototypage).
- Dans certains cas, forcer à typer un programme peut contraindre, voire **empêcher** d'écrire un code correct.

Programmation dirigée par les fonctions

Deux styles de programmation centrés autour des ensembles de fonctions :

Définition (Programmation dirigée par les données)

La **programmmation dirigée par les données** (*data-driven programmming*) est un style de programmation construit sur des familles de fonctions que l'on compose pour organiser dynamiquement le flot de contrôle.

Définition (Programmation événementielle)

La **programmation événementielle** (*event-driven programming*) est un style de programmation manipulant le flot de contrôle autour des événements.

- Tous deux considèrent les traitements comme des données à part entière.
- Les fonctions, profitant de la 1ère classe, s'insèrent naturellement dans de tels styles de programmation.

Exemple : compteur (1/2)

Considérons l'exemple d'une page permettant de manipuler un compteur :

```
import { h, text, app } from "https://unpkg.com/hyperapp";

const initialCpt = { count: 0 };                                  // Initial state

const actionsCpt = {                                              // Possible actions
    down: (state, event) ⇒ ({ ...state, count: state.count - 1 }),
    up:   (state, event) ⇒ ({ ...state, count: state.count + 1 }),
};

const viewCpt = (state) ⇒                                         // Page view
    h("main", {}, [
        h("h1", {}, text(`Counter : ${state.count}`)),
        h("button", { onclick: actionsCpt.up },   text("+1")),
        h("button", { onclick: actionsCpt.down }, text("-1")),
    ]);

app({ init: initialCpt,                                           // Tying everything together
      view: viewCpt,
      node: document.getElementById("app") });
```

(exemple utilisant la bibliothèque hyperapp et adapté depuis cette page)

Exemple : compteur (2/2)

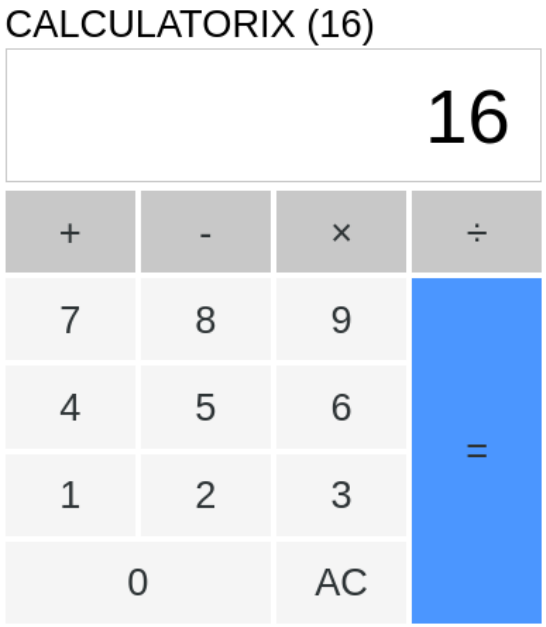
Il s'agit d'un exemple de programmation **événementielle** :

- Le modèle de la page est décrit sous la forme d'un **état** global.
- Les traitements sont organisés dans une liste `actionsCpt` :
  - Chaque traitement est rattaché à un événement, et opère une **transformation** de l'état.
  - Étendre ou contraindre les actions possibles peut se faire dynamiquement en modifiant `actionsCpt`.
- La vue est **composée** (par une fonction) à partir de l'état global et associe les événements aux actions.

Exemple : application web

Hyperapp avec un exemple d'application un peu plus complexe (Lien) :

```
const computer = {
  "+": (a, b) ⇒ a + b,
  "-": (a, b) ⇒ a - b,
  "×": (a, b) ⇒ a * b,
  "÷": (a, b) ⇒ a / b
}

const initialState = {
  fn: "",
  carry: 0,
  value: 0,
  hasCarry: false
}
```



Exemple : interpréteur BASIC (1/3)

Considérons l'exemple d'un langage de programmation comme le BASIC :

```
// A program in BASIC
10 LET I = 1

20 LET S = 1

30 LET I = I + 1



40 LET S = S * I



50 IF (I = 5) THEN GOTO 70



60 GOTO 30

70 END
```

```
const prog = [  // The equivalent in Ecmascript as an array of lines
    { num: 10,
      instr: { kind: "instrLet", id: "i",
               expr: { kind: "exprInt", int: 1}} },
    { num: 20,
      instr: { kind: "instrLet", id: "s",
               expr: { kind: "exprInt", int: 1}} },
    { num: 30,
      instr: { kind: "instrLet", id: "i",
               expr: { kind: "exprBin", op: { kind: "opPlus"},
                       lhs: { kind: "exprVar", var: "i"},
                       rhs: { kind: "exprInt", int: 1} }} },
    { num: 40,
      instr: { kind: "instrLet", id: "s",
               expr: { kind: "exprBin", op: { kind: "opMult"},
                       lhs: { kind: "exprVar", var: "i"},
                       rhs: { kind: "exprVar", var: "s"} }} },
    { num: 50,
      instr: { kind: "instrJumpIf",
               cond: { kind: "exprBin", op: { kind: "opEqual" },
                       lhs: { kind: "exprVar", var: "i"},
                       rhs:  {kind: "exprInt", int: 5}},
               num: 70} },
    { num: 60,
      instr: { kind: "instrGoto", num: 30} },
    { num: 70,
      instr: { kind: "instrEnd" } }                              ]
```

Exemple : interpréteur BASIC (2/3)

Comment faire pour interpréter un tel programme ?

- Partir d'un état gérant la ligne courante et l'environnement :

```
const initialState = { env: { },     // The variables in the environment
                       num: 10   }; // The current line number
```

- Gérer une instruction `goto` :

```
const instrGoto = (state) ⇒ (line) ⇒ { env: state.env, num: line.num };
```

- Gérer une instruction `let` :

```
const instrLet = (state) ⇒ (line) ⇒ {
            env:  { ...state.env,  // Update the environment
                    line.instr.id: evalExpr(line.instr.expr, state.env) },
            num: state.num + 10 };
```

Exemple : interpréteur BASIC (3/3)

- Il s'agit ici d'un exemple de programmation **dirigée par les données**.
- Les actions du langage peuvent être stockées dans un dictionnaire :

```
const actions = {
  instrGoto:   (state) ⇒ (line) ⇒ { ... },
  instrLet:    (state) ⇒ (line) ⇒ { ... },
  instrJumpIf: (state) ⇒ (line) ⇒ { ... },
};
```

- Les données ici sont le programme à interpréter sous la forme d'une liste d'instructions, et la liste des actions appliquables pour chaque instruction.

Possibilités

- Lister les actions disponibles, en ajouter ou en enlever dynamiquement.
- Tester le comportement d'une seule instruction facilement.

Intérêts de la prog. dirigée par les données

- Penser les programmes comme des **ensembles de comportements** avec peu de dépendances, appliquables et composables à la demande.
- De tels programmmes permettent d'**agir dynamiquement** sur ces comportements : les lister, en ajouter ou en enlever ...
- Hélas, si la liste des comportements accessibles peut varier dynamiquement, la vérification du code est rendue plus difficile.

Contrôle de l'évaluation

Définition (Contrôle de l'évaluation)

Le **contrôle de l'évaluation** (*evaluation control*) est une technique de programmation consistant à maîtriser le flot de contrôle en le stoppant et le reprenant en fonction des besoins du programme.

Permet d'envisager un nombre considérable de techniques :

- Le **déboguage** de programmes profite naturellement de pouvoir arrêter et reprendre le cours d'une exécution.
- De nombreux problèmes pour lesquels la **terminaison** n'est pas évidente peuvent aussi bénéficier du contrôle de leur exécution :
  - les explorations de grands domaines (ex. : arbres de jeu),
  - les algorithmes non-garantis de terminer (ex. : calcul de limite),
  - les programmes dépendant de facteurs externes (ex. : fetch url) ...

Congélation, décongélation

Les fonctions se prêtent bien au jeu du contrôle de l'évaluation :

Définition (Congélation)

Congeler (*to freeze*) une expression `e` consiste à construire une fonction renvoyant `e` lorsqu'elle est appliquée.

Exemple

L'expression `fact(10**10)` est un exemple de calcul extrêmement long.
La fonction `() ⇒ fact(10**10)` représente sa version congelée.

Définition (Décongélation)

Décongeler (*to thaw*) une expression congelée consiste à appliquer la fonction pour récupérer la valeur enfermée à l'intérieur.

Exemples d'expressions congélées

- Une expression congelée peut être décongelée au moment voulu :

```
let longComputation = () ⇒ fact(10**10);
// Do something else more interesting
let result = longComputation();   // → a big number
```

- Il est possible de congeler n'importe quelle expression :

```
let printingComputation = () ⇒ { console.log('Now'); };
printingComputation(); // Log : Now

let failingComputation = () ⇒ { throw new Error("Yikes"); };
failingComputation();   // Uncaught Error : Yikes
```

- Une valeur congelée peut même ne jamais être décongelée si jamais l'utilité ne se présente pas.

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function fact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return fact(n-1, n*r);
}
```

```
fact(5, 1);
// Log : fact(5, 1)
// Log : fact(4, 5)
// Log : fact(3, 20)
// Log : fact(2, 60)
// Log : fact(1, 120)
// → 120
```

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function frozenFact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return () ⇒ frozenFact(n-1, n*r);
}
```

Exemple de congélation récursive

Considérons le programme calculant la fonction factorielle :

```
function frozenFact(n, r) {
  console.log(`fact(${n}, ${r})`);
  if (n <= 1)
    return r;
  else
    return () ⇒ frozenFact(n-1, n*r);
}
```

```
let s1 = frozenFact(5, 1);
// Log : fact(5, 1), → Function
let s2 = s1();
// Log : fact(4, 5), → Function
let s3 = s2();
// Log : fact(3, 20), → Function
let s4 = s3();
// Log : fact(2, 60), → Function
let s5 = s4();
// Log : fact(1, 120), → 120
```

- Permet d'obtenir une fonction réalisant ses calculs **pas à pas**.
- L'écriture sous forme récursive terminale est fondamentale ici.

Application : les trampolines

Un **trampoline** est une fonction enlevant toutes les couches de congélation d'une valeur congelée.

```
function trampoline(v) {
    let res = v;
    while (res instanceof Function) { res = res(); }
    return res;
}
```

- Les trampolines peuvent être utilisés pour évaluer les fonctions dans le tas plutôt que dans la pile.
- Ils permettent ainsi d'écrire des fonctions récursives débarrassées des limites de taille de la pile (**stack overflow**).

```
trampoline(frozenFact(11000n, 1n)) // → some big number
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sum_squares(3, 5 - 1)   →   ???
```

On distingue :

- l'évaluation par valeur ;
- l'évaluation par nom ;
- l'évaluation paresseuse.

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation par valeur)

L'**évaluation par valeur** (*call-by-value*) est une technique d'évaluation d'une expression $f(x_1, \ldots, x_n)$ consistant à évaluer d'abord récursivement les paramètres x_i avant de les transmettre à la fonction f.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   sum_squares(3, 4)        // eval params
                       →   square(3) + square(4)    // eval sumSquares
                       →   3*3 + 4*4                // eval params of '+'
                       →   9 + 16                   // ...
                       →   25                       // eval '+'
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation par nom)

L'**évaluation par nom** (*call-by-name*) est une technique d'évaluation d'une expression $f(x_1, \ldots, x_n)$ consistant à substituer les x_i sans les évaluer dans le corps de la fonction f avant d'évaluer f.

```
let square     = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   square(3) + square(5-1)  // subs sumSquares
                       →   3*3 + (5-1)*(5-1)        // subs square (!)
                       →   3*3 + 4*4                // eval '-'
                       →   9 + 16                   // eval '*' twice
                       →   25
```

Stratégies d'évaluation

Contrôler l'évaluation permet diverses stratégies . . . d'évaluation.

Définition (Évaluation paresseuse)

L'**évaluation paresseuse** (*lazy evaluation* ou *call-by-need*) est une technique d'évaluation améliorant l'évaluation par nom en mémorisant les calculs réalisés afin de ne pas les dupliquer.

```
let square    = (x) ⇒ x*x;
let sumSquares = (a, b) ⇒ square(a) + square(b);
```

```
sumSquares(3, 5 - 1)   →   square(3) + square(5-1)   // eval sumSquares
                       →   3*3 + x*x                 // eval x → 5 − 1 → 4
                       →   9 + 4*4
                       →   9 + 16
                       →   25
```

Et si on était tout le temps paresseux ?

- Rappel : si une expression est **pure**, le résultat de son évaluation ne dépend pas du moment où elle est évaluée.
- La programmation paresseuse ne peut s'appliquer facilement que dans les calculs **sans** effets de bords.
  Dans les autres cas, il faut faire attention à l'ordre des calculs.
- Quand elle s'applique, elle permet de ne réaliser les calculs qu'au moment où ils servent, et même de les éviter s'ils ne servent pas.

Haskell applique une stratégie d'évaluation paresseuse par défaut.

```
isPrime n = length [ d | d ← [1..(n `div` 2)], n `rem` d == 0 ] == 1
primes = filter isPrime $ [1..] -- filter `isPrime` on the list of all integers
primes                          -- → [2,3,5,7,11,13,17,19,23...], never stops
```

```haskell
head []     = undefined -- or some kind of error
head (x:xs) = x

insert x []     = [x]
insert x (y:ys) = if (x<y) then (x:y:ys) else (y:insert x ys)

insert_sort []     = []
insert_sort [x]    = [x]
insert_sort (x:xs) = insert x (insert_sort xs)

min xs = head (insert_sort xs)    -- Compute the minimum element of a list
```

```haskell
min [3,2,1]
    ⟶ head (insert_sort [3,2,1])
    ⟶ head (insert 3 (insert_sort [2,1]))
    ⟶ head (insert 3 (insert 2 (insert_sort [1])))
    ⟶ head (insert 3 (insert 2 [1]))
    ⟶ head (insert 3 (if (2<1) then (2:...) else (1:...))) -- '...' stand for
    ⟶ head (insert 3 (1:...))                               -- unevaluated
    ⟶ head (if (3<1) then (3:...) else (1:...))             -- computations
    ⟶ head (1:...)
    ⟶ 1
```

Structures paresseuses (1/3)

Les valeurs congelées peuvent être stockées dans des structures de données.

Exemple

Une liste peut voir sa tête et sa queue congelées.
Les décongeler permet de déplier la liste au fur et à mesure des calculs.

Structures paresseuses (2/3)

Dans l'exemple suivant, la liste est congelée à partir d'un certain point :

```
function frozenCoins() {
    const aFlip = (Math.random() > 0.5) ? 'heads' : 'tails';
    return cons(aFlip, frozenCoins);          // the tail of the list is frozen
}

function flipACoin(c) {
    let pointer = c;
    while (typeof pointer.cdr === 'object')   // go to the end of the list
        pointer = pointer.cdr;
    pointer.cdr = pointer.cdr();              // replace its tail
}
```

```
let c = frozenCoins(); // { 'tails', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', 'heads', [Function: frozenCoins] }
flipACoin(c);          // { 'tails', 'heads', 'heads', 'tails', [Function: frozenCoins] }
```

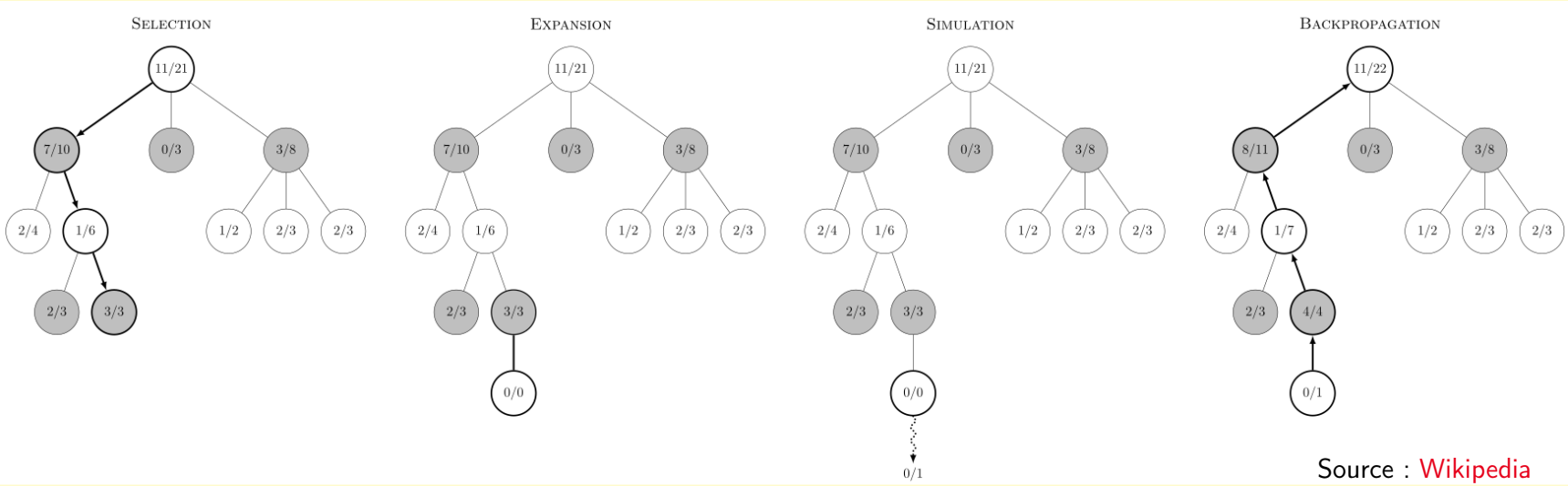
Structures paresseuses (3/3)

Les structures paresseuses permettent d'envisager :

- de construire des structures de taille quelconque, voire infinie ;
- de construire des structures qui se développent selon un algorithme.

Exemple : Monte-Carlo tree search

Exploration de l'arbre des parties d'un jeu pour évaluer les coups optimaux :



Source : Wikipedia

Promesses (1/3)

Les calculs congelés ont le bon goût de pouvoir se composer sans s'évaluer :

- Supposons disposer d'une valeur congelée : `let fznInt = () ⇒ fact(10);`
- Supposons vouloir la transformer par `Math.sqrt` sans la décongeler.

Pour cela, il suffit simplement de construire la valeur congelée suivante :

```
() ⇒ Math.sqrt(fznInt())
```

En généralisant, on obtient une fonction agissant sur les valeurs congelées :

```
function fznSquare(fznInt) {                    // frozen value → frozen value
    return () ⇒ Math.sqrt(fznInt());
}
```

```
let fznInt = () ⇒ fact(10);       // → [Function] (frozen value)
let result = fznSquare(fznInt);   // → [Function] (frozen value)
result();                         // → 1904.94
```

Promesses (2/3)

Définition (Promesse)

Une **promesse** (*promise*) Ecmascript est la réalisation d'un calcul asynchrone.

On la construit à partir d'une fonction à deux paramètres fonctionnels :

- `resolve`, qui est appelée sur la valeur renvoyée par la promesse,
- `reject`, qui est appelée sur la valeur d'erreur en cas d'erreur.

```
const promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve('hello');
    }, 10000);
 })                              // Promise returning 'hello' after 10 seconds
```

- A la différence d'une valeur congelée, une promesse commence à être évaluée dès sa définition

Promesses (3/3)

Comme les promesses sont des calculs asynchrones, il est là aussi intéressant de pouvoir les composer :

- la méthode `then` permet de spécifier que faire à la fin du calcul :

```
const promiseCpx = promise
      .then((value) ⇒ { return value.toUpperCase(); })
      .then((value) ⇒ { return `${value} - ${value}` });
const result = await promiseCpx; // → "HELLO - HELLO"
```

- la méthode `catch` permet de spécifier que faire en cas d'erreur.

```
promise.catch((err) ⇒ { console.log(err); });
```

Idée

Composer les promesses (ou les valeurs congelées), c'est construire un calcul en composant des fonctions.

Conclusion

- L'étude de la programmation fonctionnelle peut s'aborder et se développer sur de nombreuses techniques.
- La maîtrise de chacune de ces techniques n'est pas évidente et dépasse le cadre de ce cours.
- L'idée centrale tient au fait que les fonctions sont des unités de code portables, réutilisables, appliquables à la demande, et permettant d'organiser dynamiquement le flot de contrôle d'un programme.
- La suite du cours portera sur l'architecture de ces ensembles de fonctions.

Modularité

Idée générale

Considérer le code non plus au niveau de la **fonction**, mais des **ensembles de fonctions**, voire des ensembles de fonctionnalités.

- Le terme utilisé pour différencier les deux est celui de **granularité**.
- Transforme les problématiques de programmation en des problématiques d'architecture.

Abtraction, Encapsulation

Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :

- **Abstraction** : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

Abtraction, Encapsulation

Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :

- **Abstraction** : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

(découpage naïf avec fermeture)

```
function make_rng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };                              Private
  return rand;                    Public
}
let rand = make_rng();
```

Abtraction, Encapsulation

Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :

- **Abstraction** : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

(découpage naïf avec fermeture)

```
function make_rng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };                              Private
  return rand;                    Public
}
let rand = make_rng();
```

⇒

(cf. Stanford JS Crypto Library)

```
class Aes;
function aes_generate() { /*...*/ }
class EllipticCurve;
function ecc_generate() { /*...*/ }
function sha256() { /*...*/ }
function sha512() { /*...*/ }          Private

function selectCypher() { /*...*/ }
function encrypt() { /*...*/ }
function decrypt() { /*...*/ }         Public
```

Spécification

Quelles peuvent être ces caractéristiques essentielles d'un programme ?
Concrètement, il faut parler de **spécification**. Il en existe plusieurs sortes :

- Spécification **fonctionnelle** :
  - "Il doit être possible d'ajouter ou d'ôter un élément de l'ensemble."
  - "Il doit être possible de tester si un élément appartient ou pas à l'ensemble."

- Spécification **non fonctionnelle** :
  - "L'implémentation doit utiliser des algorithmes de complexité linéaire."
  - "L'implémentation doit pouvoir gérer des ensembles de taille arbitraire."

Spécification

Quelles peuvent être ces caractéristiques essentielles d'un programme ?
Concrètement, il faut parler de **spécification**. Il en existe plusieurs sortes :

- Spécification d'**interface** ou de **type** :

```
;; T is the type of the
;; elements inside the set.
newtype set[T];
```

```
set_empty    : set[T]
set_is_empty : set[T] → boolean
set_add      : set[T]*T → set[T]
set_remove   : set[T]*T → set[T]
set_find     : set[T]*T → boolean
```

- Spécification **formelle** :

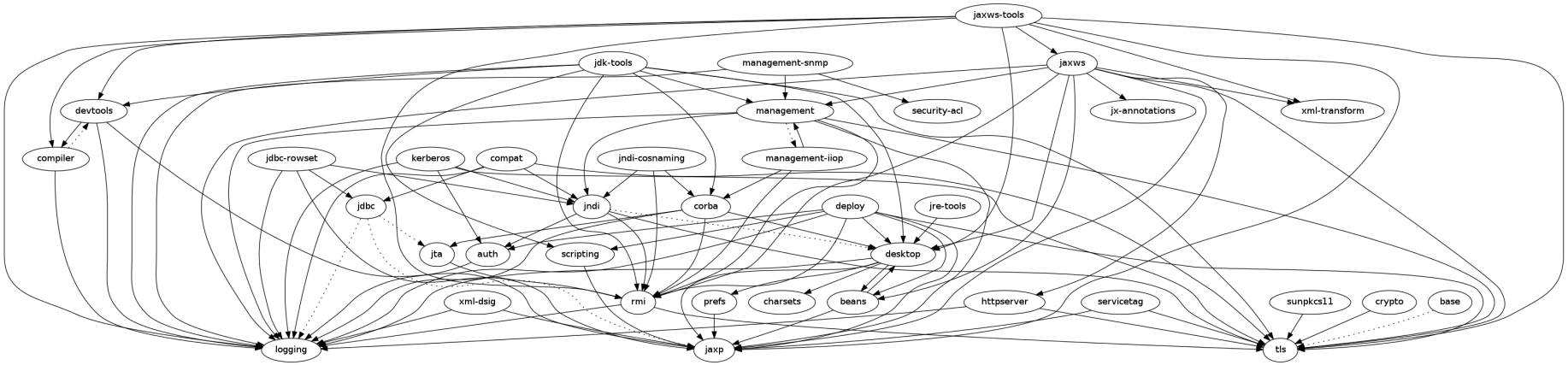
```
set_is_empty(set_empty)        = true
set_is_empty(set_add(s,i))     = false

set_find(set_empty, i)         = false
set_find(set_add(s, i), i)     = true
set_find(set_add(s, i), i')    = set_find(s, i')
```

Modularité

Définition (Modularité)

Propriété logicielle qualifiant le fait de décomposer un programme en un ensemble de composants de manière à ce que :

- **Cohésion** : chaque composant constitue une unité de code cohérente ;
- **Couplage** : les dépendances entre composants sont faibles.



Graphe de dépendances de l'OpenJDK
Source : Project Jigsaw

Module

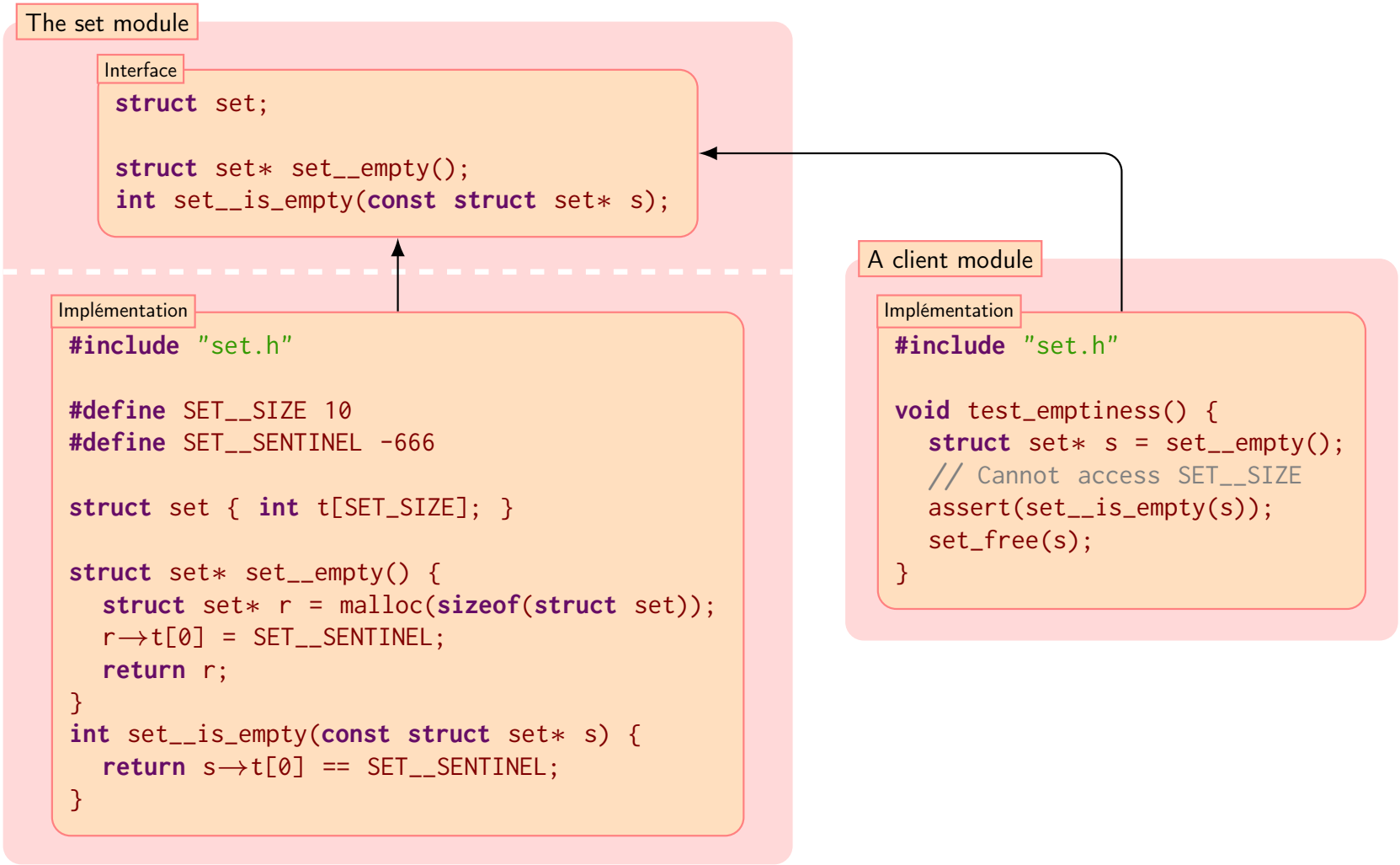
Définition (Module)

Un **module** est une construction du langage représentant une unité de code (types, valeurs, fonctions, et toute forme d'expression autorisée par le langage) et satisfaisant :

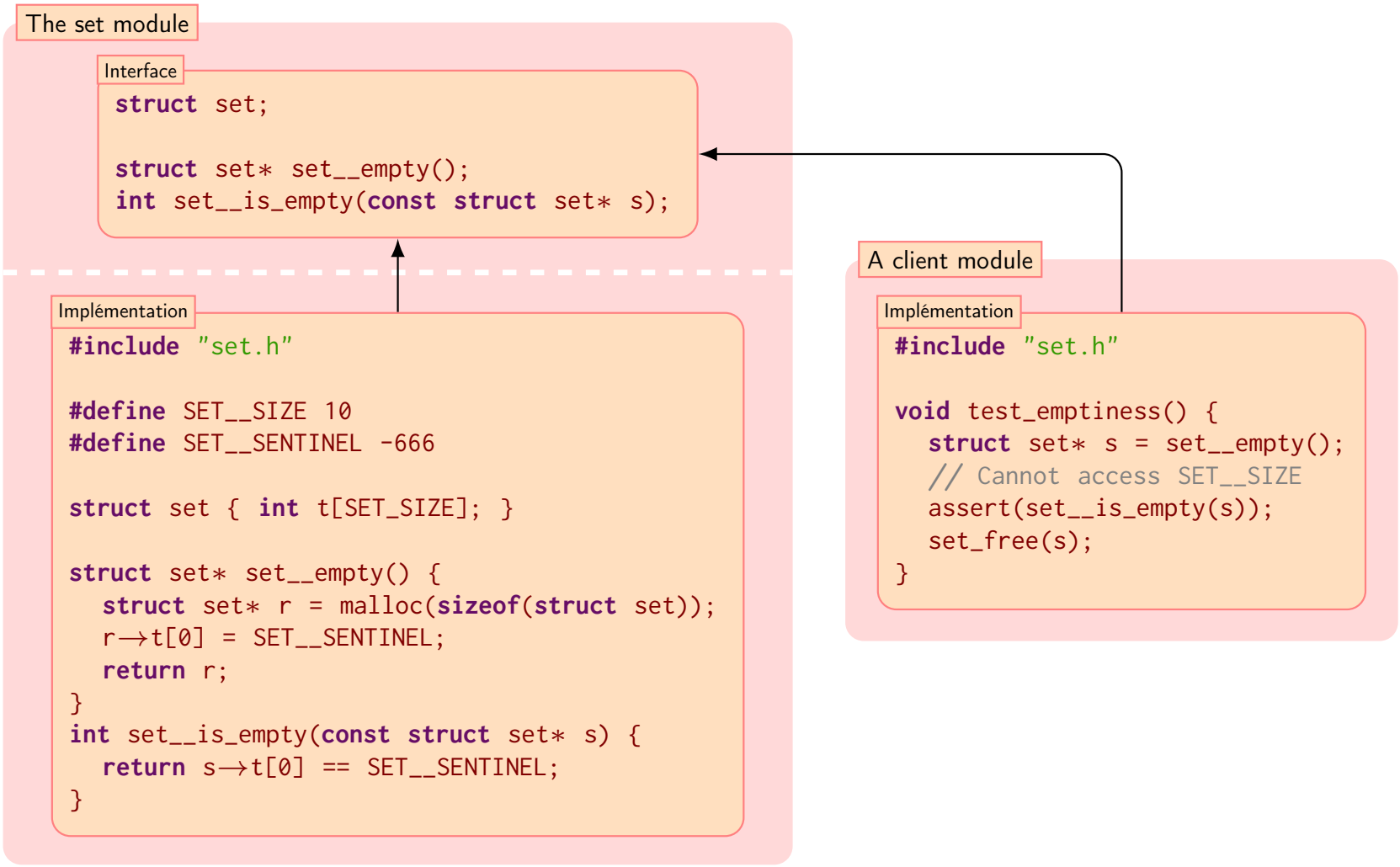
- **Interface** : un module peut fournir ou requérir un ensemble de composants de manière publique ;
- **Encapsulation** : un module peut masquer ou rendre abstrait une partie de ses composants ;

- Des ensembles de modules peuvent se connecter selon le graphe des dépendances induites par leurs interfaces ;
- Un module ne devrait **dépendre que des interfaces** de ses dépendances (et non pas de leur contenu concret).

Exemple : des modules en C



Exemple : des modules en C



Interface ✓ Encapsulation ✓ Indépendence ✓

Exemple : modules et TADs

Un exemple classique de module est le suivant :

Définition (Type abstrait de données)

Un **type abstrait de données** (*abstract data type*) consiste en :

- un ensemble de valeurs ;
- une interface / représentation abstraite de cet ensemble de valeurs (e.g sous la forme d'une liste des opérations qu'on peut leur appliquer) ;
- une ou plusieurs implémentations de cette interface.

L'abstraction permet ici :

- de proposer plusieurs implémentations **différentes** et dans l'idéal **substituables** du type de données ;
- d'autoriser différents clients à manipuler le type de données en restant **indépendant** de la représentation concrète.

Modules et modules . . .

Selon les langages de programmation, les modules prennent des formes variées et satisfont des propriétés différentes :

- Les interfaces peuvent permettre différents niveaux de **vérification** : uniquement les noms (Racket, Python), ou les types (C, OCaml)
- L'**encapsulation** peut ne pas exister (cf. Python module privates) ;
- Les interfaces sont parfois attachées au module (Racket, Haskell) ou sont des éléments **indépendants** (C, OCaml).

Pour la suite, nous allons examiner les possibilités modulaires du langage Ecmascript.

Ecmascript et modules

Les modules Ecmascript ont une histoire désordonnée :

- Originellement, la norme ne contenait pas de spécification pour les couches modulaires.
- A partir de 2009, plusieurs entités ont proposé des systèmes de modules différents : **CommonJS**, AMD, RequireJS ...
- La norme Ecmascript 6 a proposé un système de module en 2015, parfois nommé **Ecmascript** modules, incompatible avec les précédents.
- Node.js utilise par défaut les modules CommonJS, en offrant un support "expérimental" des modules Ecmascript (depuis 2015 !).

Développement “sans modules”

Principe

Le code est écrit comme un seul bloc, avec de faibles spécifications.

```
function cons(hd, tl)      { return { car: hd, cdr: tl }; }
function set__empty()      { return undefined; }
function set__is_empty(s)  { return s === undefined; }
function set__add(x, s)    { return cons(x, s); }
function set__find(x, s)   { return (s === undefined) ? false :
                                (x === s.car || set__find(x, s.cdr)); }
set__is_empty(set__empty());                  // → true
set__find(1, set__add(1, set__empty())); // → true
```

Se marie bien avec :

- REPL (simplifie le développement incrémental)
- Système de type dynamique (retarde la vérification)

Langages adaptés : Lisp, Scheme, Javascript, Racket, Python, Ruby

Limites du “sans modules”

Une telle pratique induit un développement **monolithique** :

- Des parties du code avec des **finalités différentes** sont mélangées dans un unique fichier (e.g. implémentation et tests) ;
- La **réutilisation** et la **modification** d'une partie du code est gênée (e.g. modification de la représentation d'un ensemble) ;
- La **vérification** du code est plus complexe (tout ou rien) ;

Une telle méthode est adaptée au **prototypage**, mais montre ses limites lors du passage à l'échelle : travail en équipe compliqué, pas de compilation séparée, pas de tests séparés . . .

Programmation modulaire

Principe de la programmation modulaire

Décomposer un code monolithique en une famille de modules cohésifs et faiblement couplés.

```
function is_set(s) { /* ... */ }
function set_add(s, e) { /* ... */ }
function set_empty() { /* ... */ }
function set_find(s, e) { /* ... */ }
function test_add() { /* ... */ }
function test_empty() { /* ... */ }
function test_find() { /* ... */ }
function set_recipes() { /* ... */ }
function find_recipe(key) { /* ... */ }
function cook_recipe(key) { /* ... */ }
```

Programmation modulaire

Principe de la programmation modulaire

Décomposer un code monolithique en une famille de modules cohésifs et faiblement couplés.

```
function is_set(s) { /* ... */ }
function set_add(s, e) { /* ... */ }
function set_empty() { /* ... */ }
function set_find(s, e) { /* ... */ }
function test_add() { /* ... */ }
function test_empty() { /* ... */ }
function test_find() { /* ... */ }
function set_recipes() { /* ... */ }
function find_recipe(key) { /* ... */ }
function cook_recipe(key) { /* ... */ }
```

Implémentation

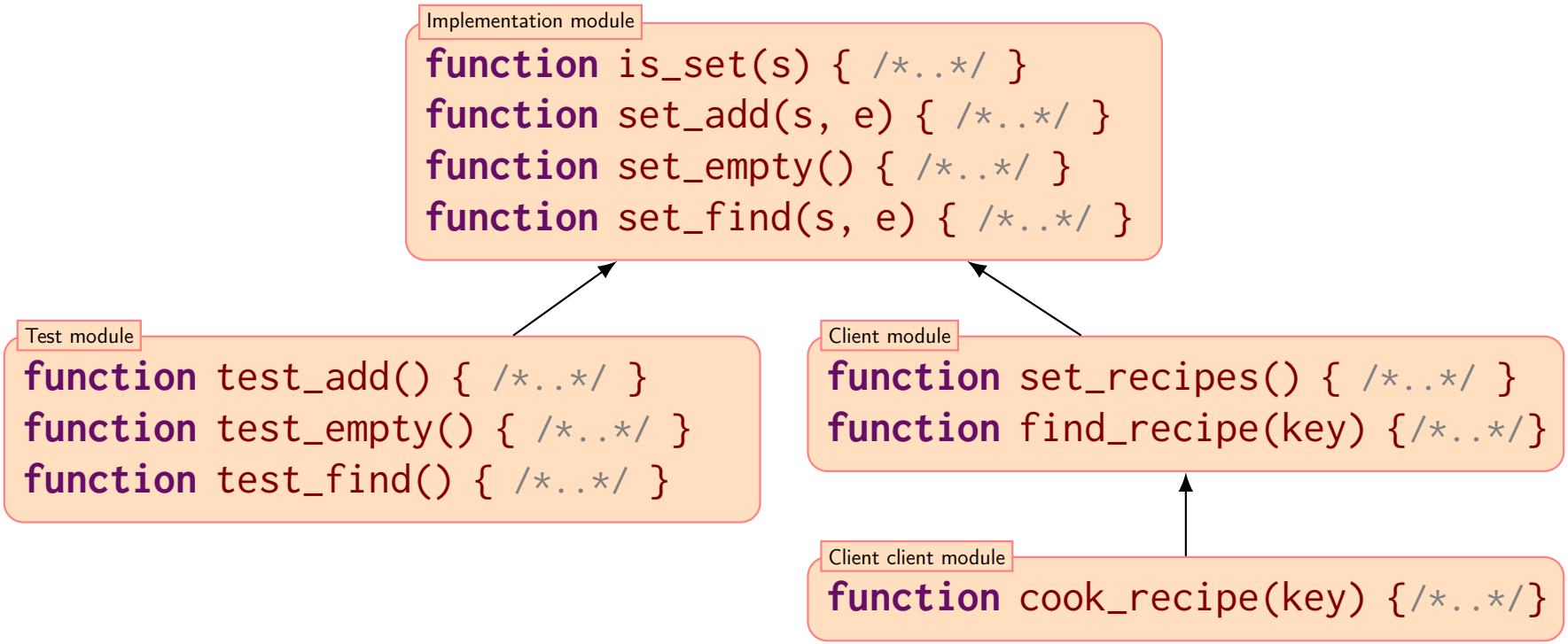
Tests

Code client

Programmation modulaire

Principe de la programmation modulaire

Décomposer un code monolithique en des ensembles de modules cohésifs et faiblement couplés.



Modules CommonJS (non utilisés dans ce cours)

Les modules CommonJS sont connectés par **require** et **exports** :

set-impl.js

```
function set_empty() { /*..*/ }
function is_empty(s) { /*..*/ }

exports.set_empty = set_empty;          // 'exports' is a dict containing
exports.is_empty  = is_empty;           // the exported identifiers
```

set-test.js

```
const Set = require("./set-impl.js");   // 'require' returns the
                                        // aforementioned dict

function test_empty() { return Set.is_empty(Set.set_empty()); }
console.log(`Empty set is empty : ${test_empty()}`);
```

set-client.js

```
const Set = require("./set-impl.js");

let set_recipes = Set.set_empty();
function add_recipe(r) { set_recipes = Set.set_add(r, set_recipes); }
```

Modules Ecmascript

Les modules Ecmascript sont connectés par **import** et **export** :

set-impl.mjs

```
function set_empty() { /*..*/ }
function is_empty(s) { /*..*/ }

export { set_empty, is_empty,          // ‘export‘ provides a list of
         set_add, set_find };          // exported identifiers
```

set-test.mjs

```
import * as Set from "./set_impl.mjs";   // ‘import‘ supplies the
                                         // requested identifiers

function test_empty() { return Set.is_empty(Set.set_empty()); }
console.log(‘Empty set is empty : ${test_empty()}‘);
```

set-client.mjs

```
import * as Set from "./set_impl.mjs";

let set_recipes = Set.set_empty();
function add_recipe(r) { set_recipes = Set.set_add(r, set_recipes); }
```

Premières conclusions

Avantages :

- Facilite la **réutilisation de code** : à la fois les tests et les clients peuvent réutiliser le même module d'implémentation ;
- Autorise des **implémentations multiples** : le client peut choisir une implémentation adéquate sans modifier son propre code ;
- Permet une **vérification** de la présence des fonctions importées et exportées.

Limitations en Ecmascript :

- Vérification réduite à la présence/absence des noms demandés ;
- Notion d'interface (d'un module) presque inexistante.

Développement de code modulaire

Recette pour construire une architecture modulaire :

1. Identifier les objets/**types de valeurs**/composants manipulés ;
   Ex. : "les ensembles d'entiers positifs"
   ⇒ leur associer un module.
2. Identifier les **opérations** sur chacun de ces types de valeurs ;
   Ex. : "`setAdd` : ajouter un élément, `setRemove` : retirer un élément"
   ⇒ contruire une interface associée.
3. Construire des **représentations concrètes** ;
   Ex. : "sentinel, dynamic, link"
   ⇒ implémenter ladite interface.

Principe général de développement modulaire

Lorsque l'on conçoit une architecture logicielle, il vaut **toujours mieux** dépendre d'une interface que d'une représentation concrète.

Même si le développement modulaire peut s'arrêter ici, il est envisageable de demander plus :

- En termes de **vérification** :
  Quitte à préciser des spécifications logicielles, existe-t'il des tactiques pour assurer que des modules vérifient une spécification ?
- En termes d'**architecture** :
  Quitte à décomposer le code en morceaux, peut-on faire évoluer ces morceaux, les augmenter, ou les remplacer facilement ?

Contrats

Définition (Contrat)

Un **contrat** est un ensemble de prédicats qui fixent les comportements des fonctions au sein d'un module (précondition, postcondition, invariants)

Comme pour les assertions, les contrats sont vérifiés **dynamiquement**.

Exemple en Racket

Documentation de la fonction `list-set` :

```
;; Updates element at index ‘pos‘ of ‘lst‘
(list-set lst pos val) → list?
  lst : list?
  pos : (and/c (>=/c 0) (</c (length lst)))
  val : any/c
```

“pos est un indice valide dans `lst`”

Utilisation incorrecte :

```
(list-set '(1 2 3) 100 4)
```

```
<=: contract violation
  expected: (and (>= 0) (< 3))
  given: 100
```

Langages adaptés : Eiffel, Racket, C# with Code Contracts

Signatures

Définition (Signature)

Une **signature** est un ensemble de définitions de types pour les identifiants exportés par un module.

Comme les types, les signatures sont vérifiées **statiquement**.

Exemple en C

Documentation de la fonction `set__find` :

```
// Returns non-zero if c belongs to se, 0 otherwise
int set__find(const struct set *se, int c);
```

Utilisation incorrecte :

```
struct set* se = set__empty();
set__find(1, se);
```

```
error: passing argument 1 of 'set__find' makes pointer from integer without a cast
expected 'const struct set *' but argument is of type 'int'
```

Langages adaptés : la plupart des langages à typage statique

Exemples de signatures

Idée

Permettre de **séparer** la signature de l'implémentation.

- **Interfaces** en Java

```
interface Set<T>
{
    Set<T>  set_add(Set<T> set, T el);
    Set<T>  set_remove(Set<T> set, T el);
    Set<T>  set_empty();
    boolean set_is_empty(Set<T> set);
    boolean set_find(Set<T> set, T el);
    int     set_length(Set<T> set);
}
```

- **Signatures de modules** en OCaml

```
module type SET = sig
  type 'a set
  val set_add      : 'a set → 'a → 'a set
  val set_remove   : 'a set → 'a → 'a set
  val set_empty    : unit   → 'a set
  val set_is_empty : 'a set → bool
  val set_find     : 'a set → 'a → bool
  val set_length   : 'a set → int
end
```

- Les signatures, comme les contrats, deviennent des éléments de l'architecture, comme les implémentations.
- L'abstraction consiste alors à dépendre uniquement des interfaces.

Conclusion sur la programmation modulaire

- Composer du code de manière modulaire est la bonne manière de travailler sur des architectures de tailles importantes.
- Les qualités d'une bonne décomposition modulaire (cohésion, faible couplage) visent à **contrôler les dépendances** entre les composants.
- La notion d'**abstraction** est au coeur du contrôle de ces dépendances.
- Une bonne décomposition permet d'envisager :
  - des propriétés de **génie logiciel** : réutilisabilité, substitutivité, extensibilité, maintenabilité ...
  - des facilités de **vérifications** des dépendances : spécifications, signatures, contrats, types ...

Conclusion générale

Il n'est pas naïf de considérer en dernier lieu de ce cours une discussion sur la notion de modularité.

- La notion de **dépendance** est un principe incontournable de programmation.
  Mieux, elle est appliquable à la fois au niveau fonctionnel (cf. pureté) et au niveau modulaire (cf. la notion de granularité) ;
- La notion de **composition** est un autre principe de programmation incontournable.
  Elle est aussi appliquable à la fois au niveau fonctionnel (cf. 1ère classe) et au niveau modulaire.

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.