# PG104 - Programmation Fonctionnelle

D. Renault

**ENSEIRB-Matmeca**

10 février 2022, v.1.1.2

# Organisation du cours

- 8 séances de cours de 2h chacune (*)
- 10 séances de TD de 2h chacune, encadrées par Myriam Desainte-Catherine, Melvin Even, Sylvain Lombardy, 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 ...).

# La 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.

# Dans les grandes lignes . . .

- 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 ce concept 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 $ax^2 + bx + c = 0$.

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

# Un petit problème simple (1/2)

**Problème (Trinôme)**

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

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

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**

A partir des relations entre des objets, on peut construire un programme.

# 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 ?

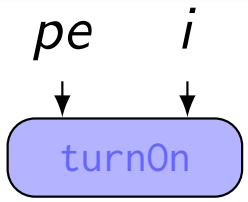
# Un problème plus complexe (2/2)

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

**Objectif**

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

# Un problème plus complexe (2/2)



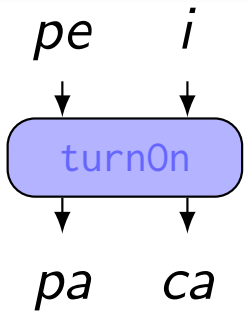

$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

**Objectif**

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

# Un problème plus complexe (2/2)

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

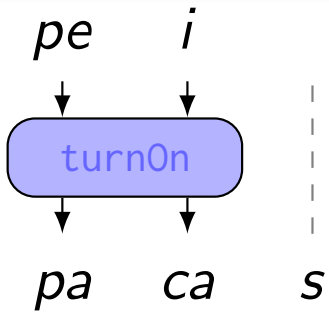



**Objectif**

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

# Un problème plus complexe (2/2)

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

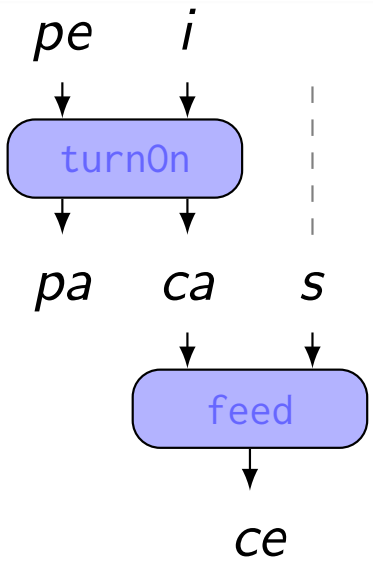



**Objectif**

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

# Un problème plus complexe (2/2)

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

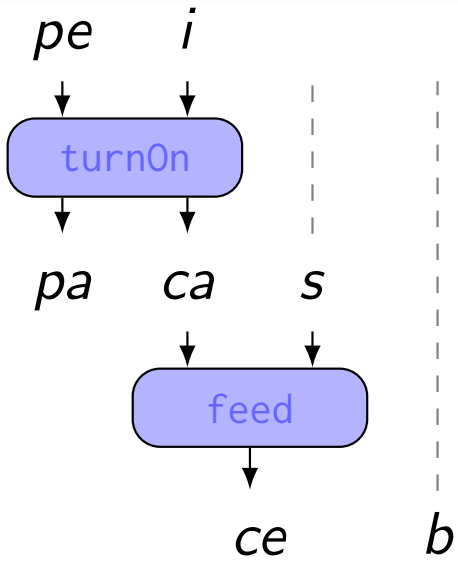



**Objectif**

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**

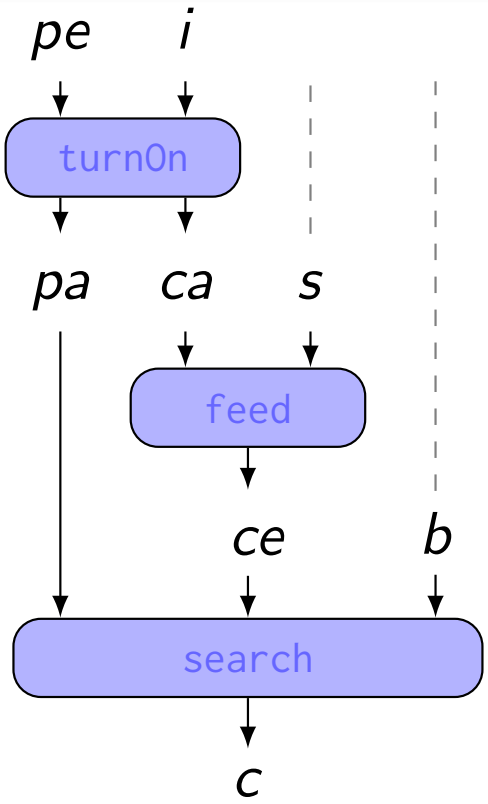
À partir de $pe$, $pf$, $i$, $b$, $s$, produire $po$.

# 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 $po$.

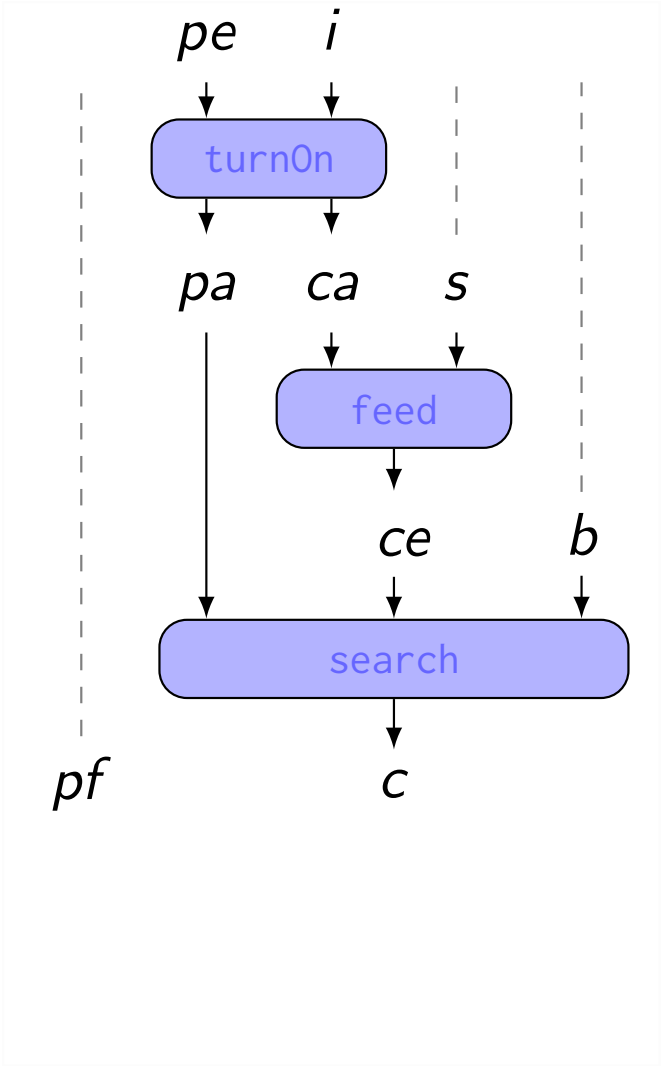


# Un problème plus complexe (2/2)

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

**Objectif**

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

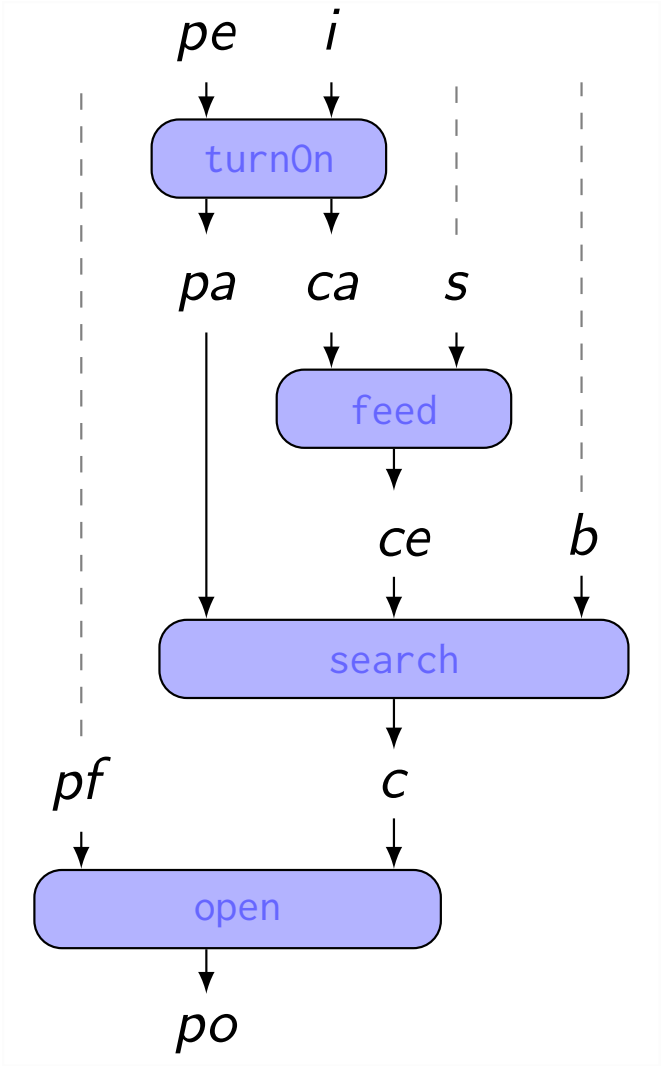


# Un problème plus complexe (2/2)

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

**Objectif**

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



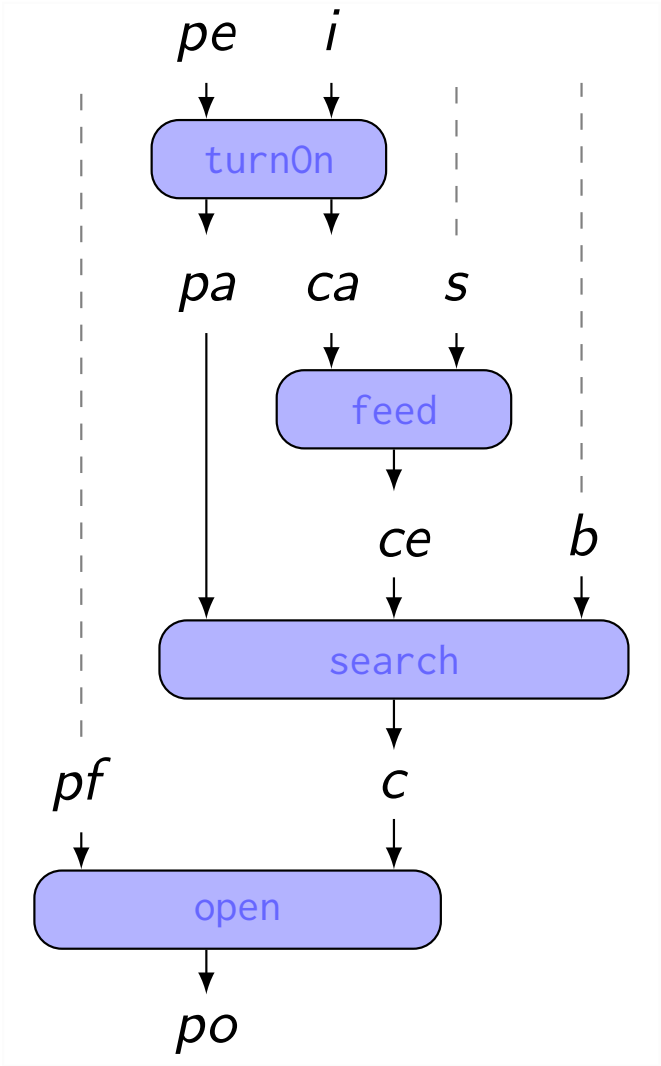
# Un problème plus complexe (2/2)

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

**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.

```
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 immutables ;
- 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, rien n'empêche de les dupliquer.

# Paradigmes de programmmation

**Définition (Paradigme)**

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 : C17 de 2018)

# Le paradigme fonctionnel

- Brique élémentaire : l'**expression**
- Idée générale :
  - partir d'un ensemble de valeurs de base (nombres . . .)
  - 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 :

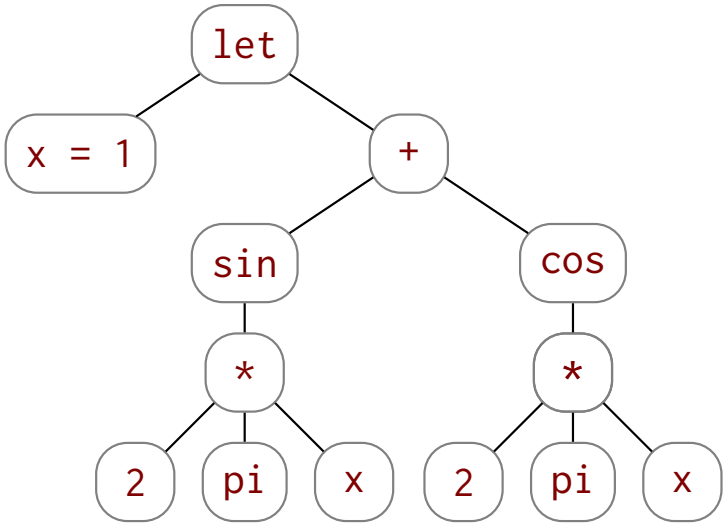
- un ensemble de valeurs $\mathcal{V}$ (entiers, chaînes de caractères, booléens ...) ;
- un ensemble $\mathcal{T}$ de moyens de composer ces valeurs (fonctions, méthodes, constructions syntaxiques particulières ...).

**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 $\mathcal{V}$.

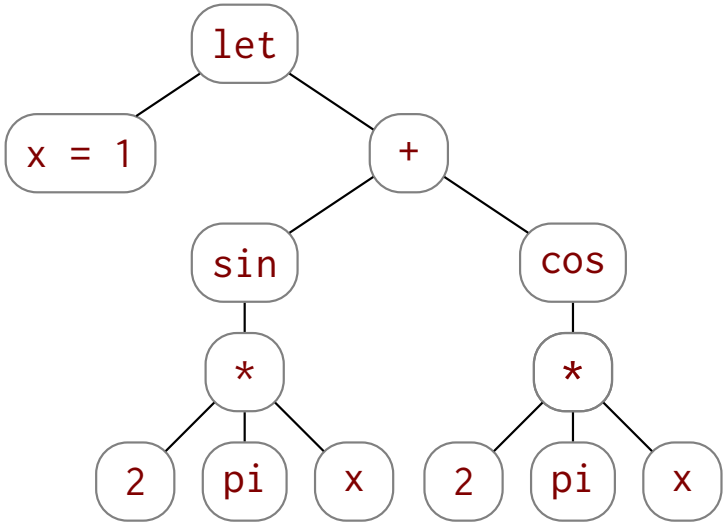
- L'expression est la représentation arborescente d'un calcul.
- **Évaluer** une expression permet d'obtenir le résultat du calcul.

# Exemples d'expressions (1/2)




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

# Exemples d'expressions (1/2)



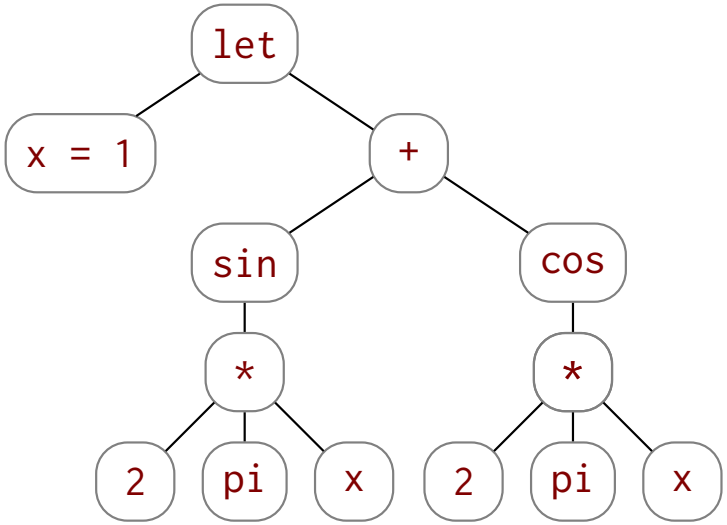

```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Haskell) :

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

-- → 0.9999999999999998
```

# Exemples d'expressions (1/2)

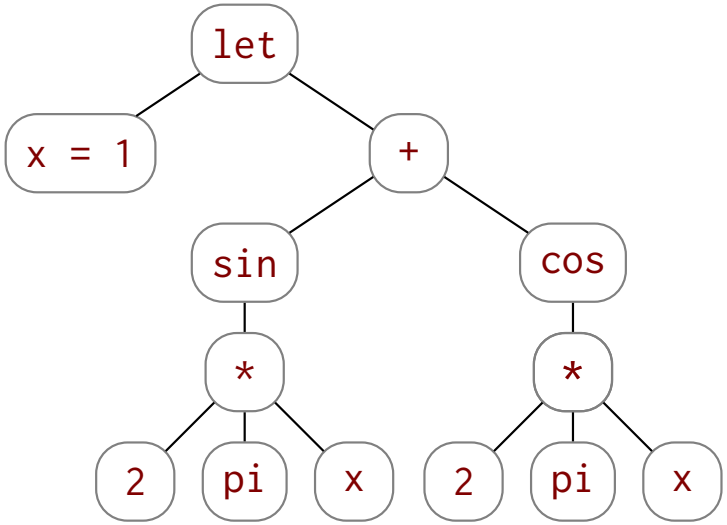



```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

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)



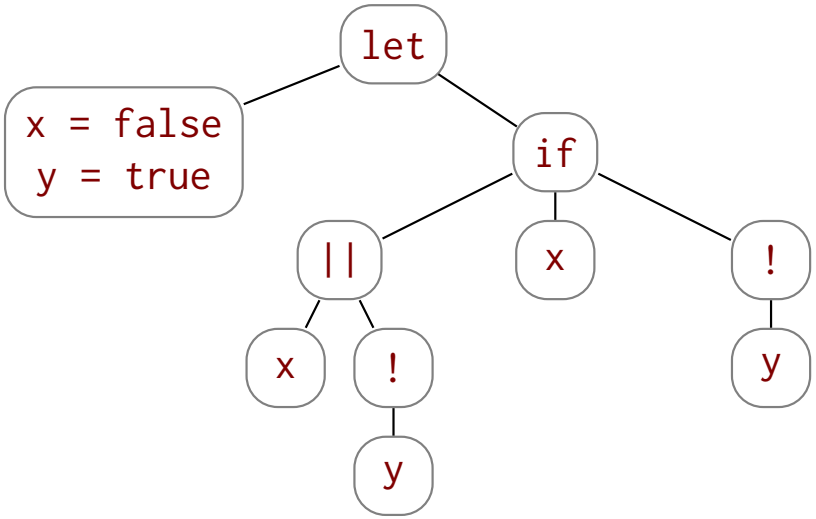

```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Javascript) :

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

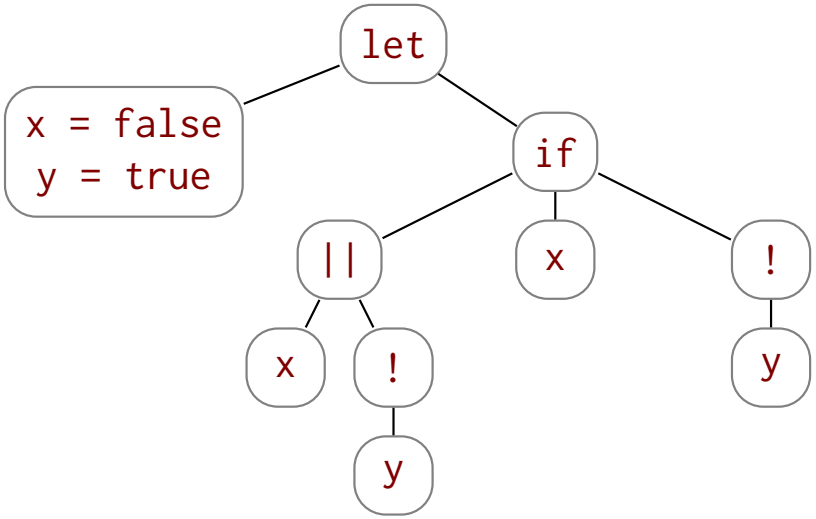
L'évaluation d'une expression se fait dans un **environnement** fournissant les valeurs des objets accessibles à un moment de l'évaluation.

# Exemples d'expressions (2/2)




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

# Exemples d'expressions (2/2)

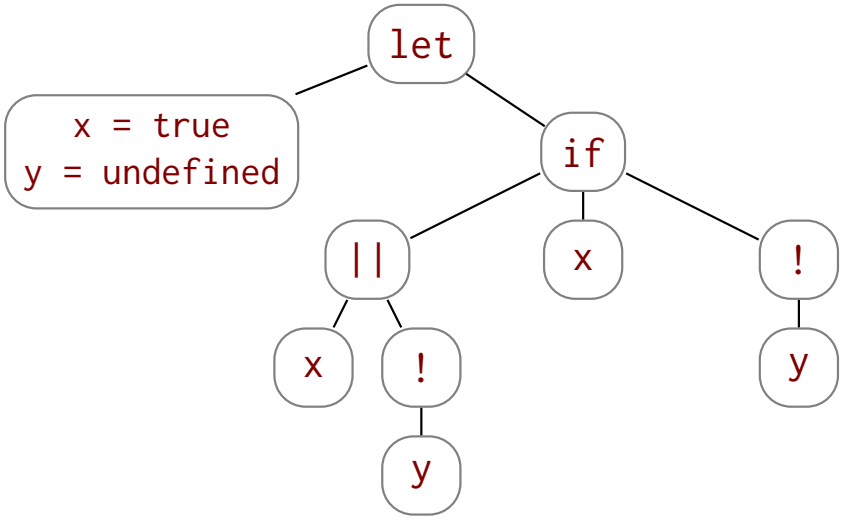



```
let x = false and y = true in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

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

# Exemples d'expressions (2/2)

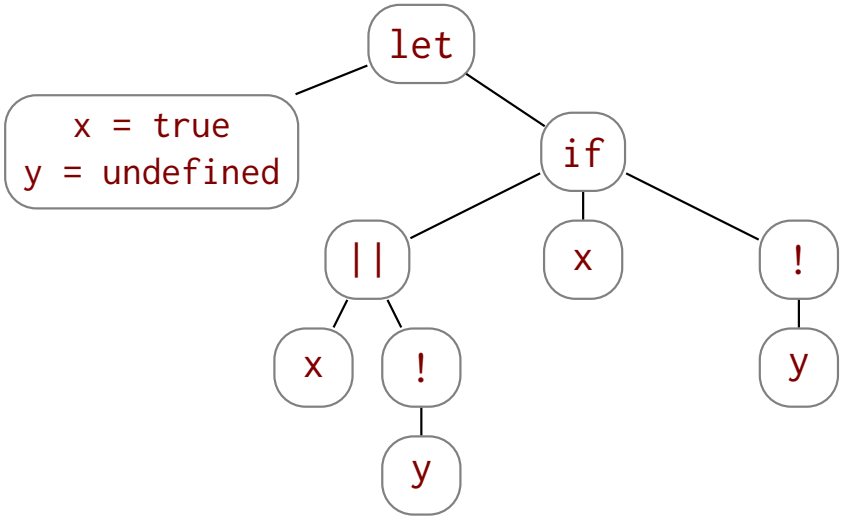



```
let x = true and y = undefined in
    if (x || !y) then x else !y
```

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`.

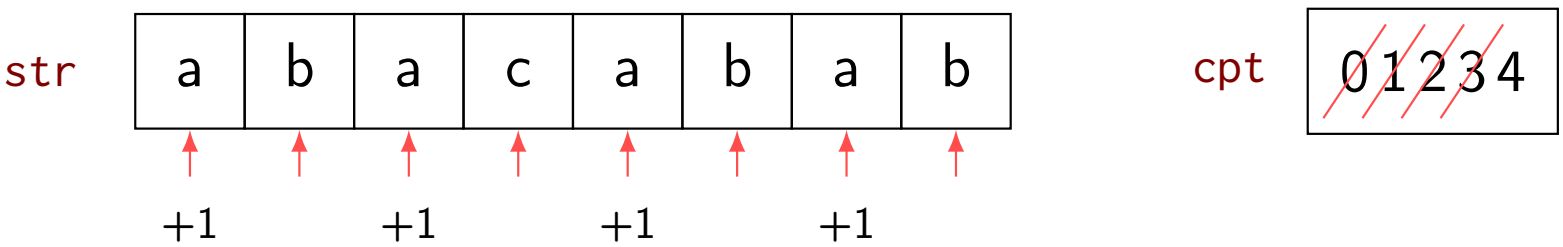
**Exemples**

```
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++)
    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`.



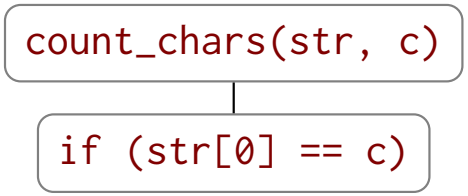
# Implémentation sous la forme d'une expression (1)

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

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

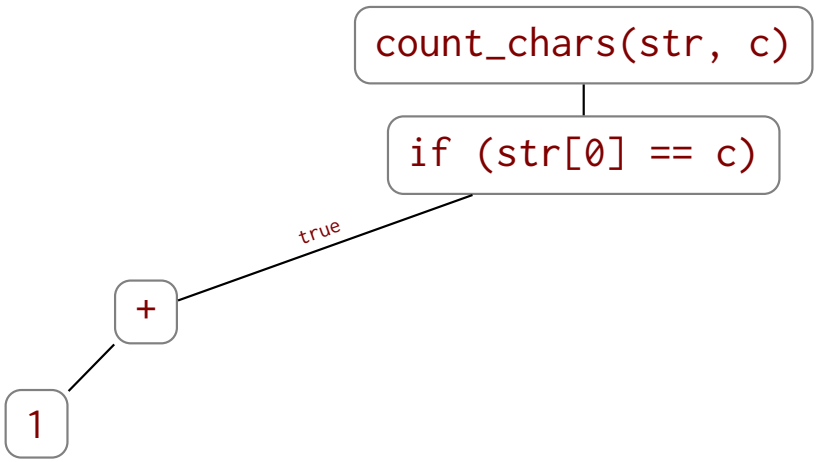
# Implémentation sous la forme d'une expression (1)

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



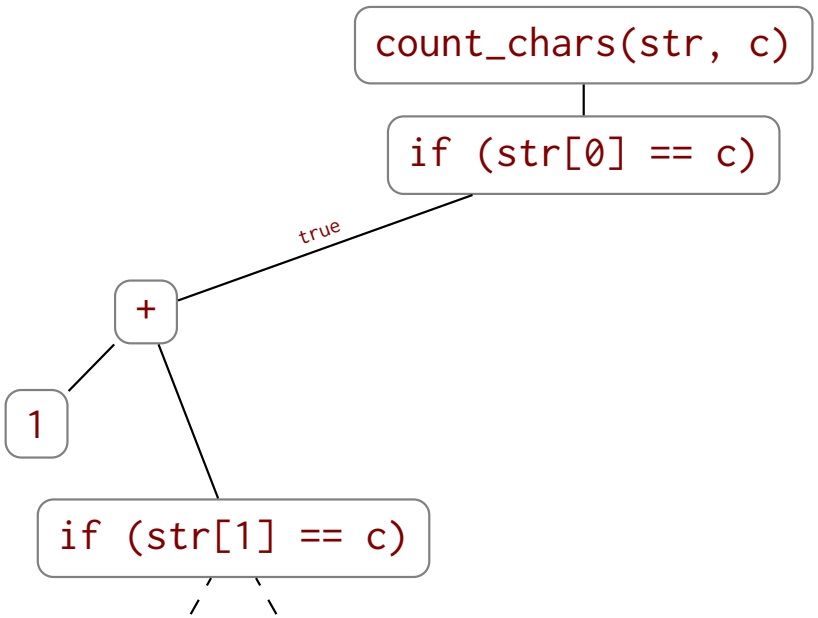
# Implémentation sous la forme d'une expression (1)

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



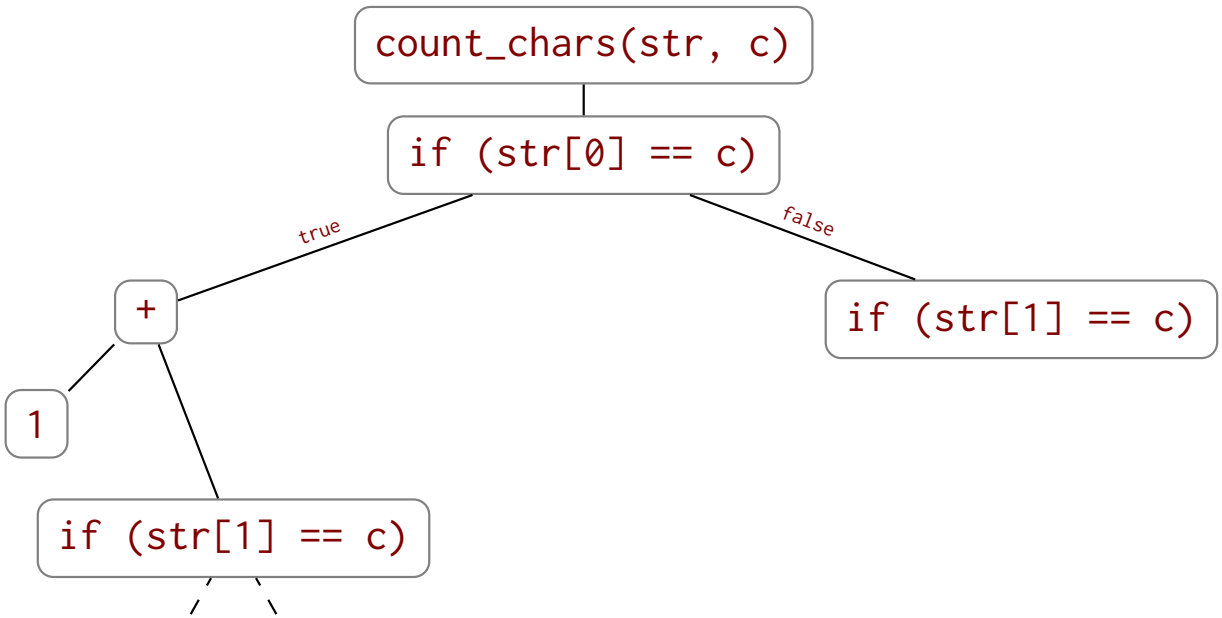
# Implémentation sous la forme d'une expression (1)

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



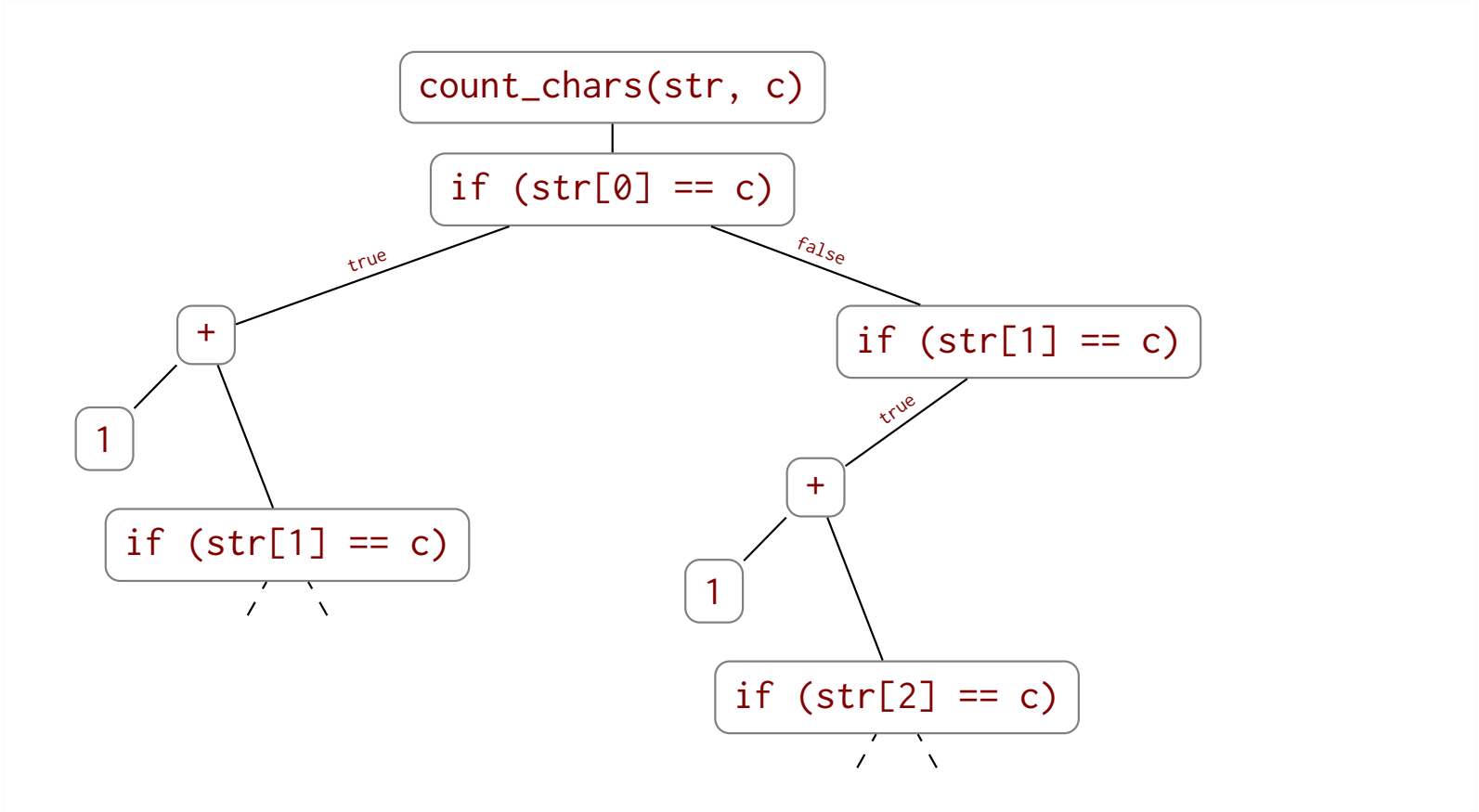
# Implémentation sous la forme d'une expression (1)

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



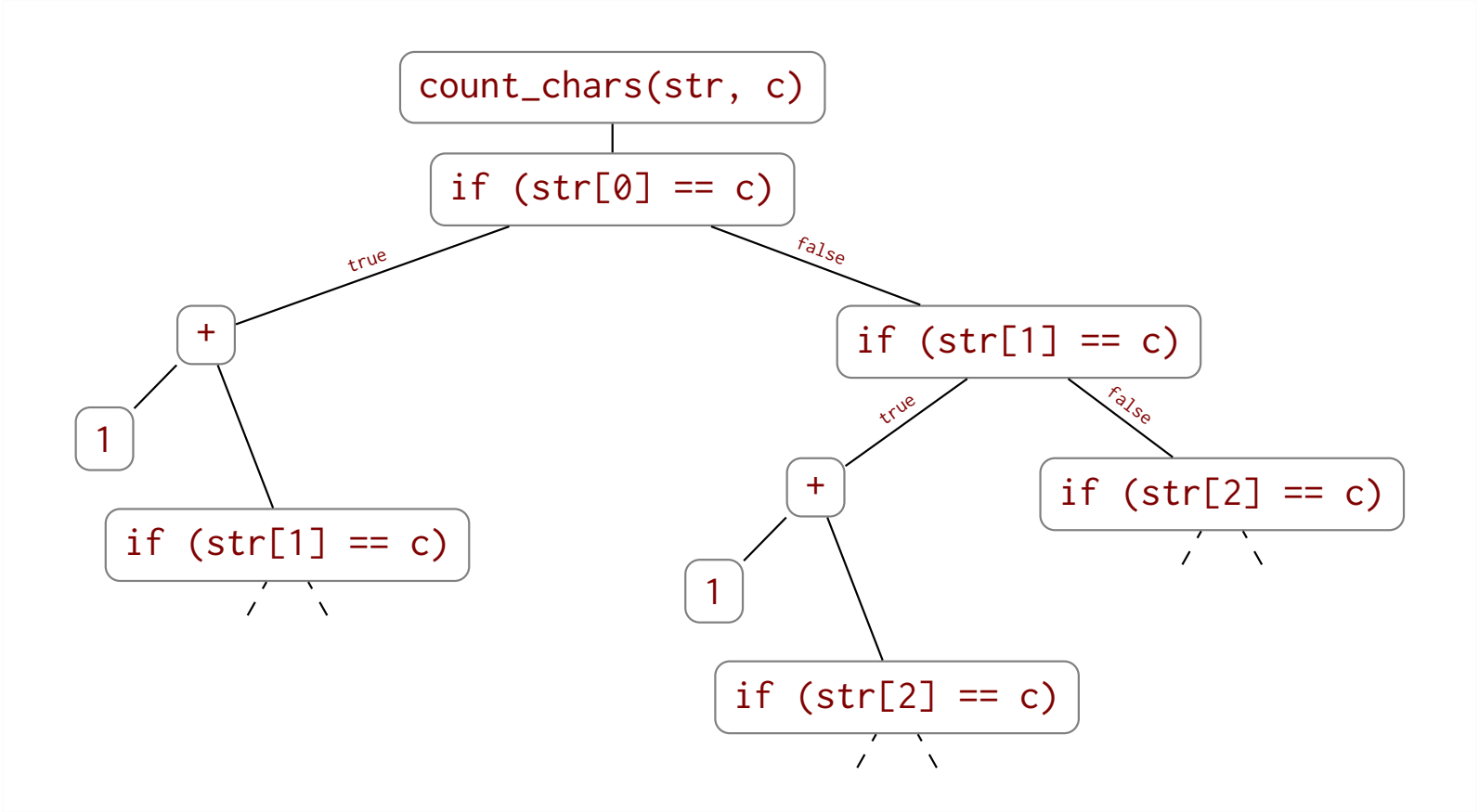
# Implémentation sous la forme d'une expression (1)

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, parce que 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` privé 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` privé de son 1er caractère.

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

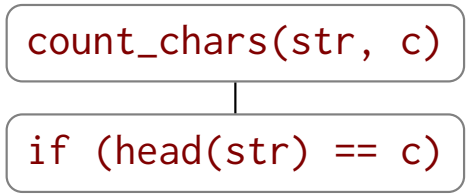
```
count_chars(str, c)
```

# 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` privé de son 1er caractère.

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

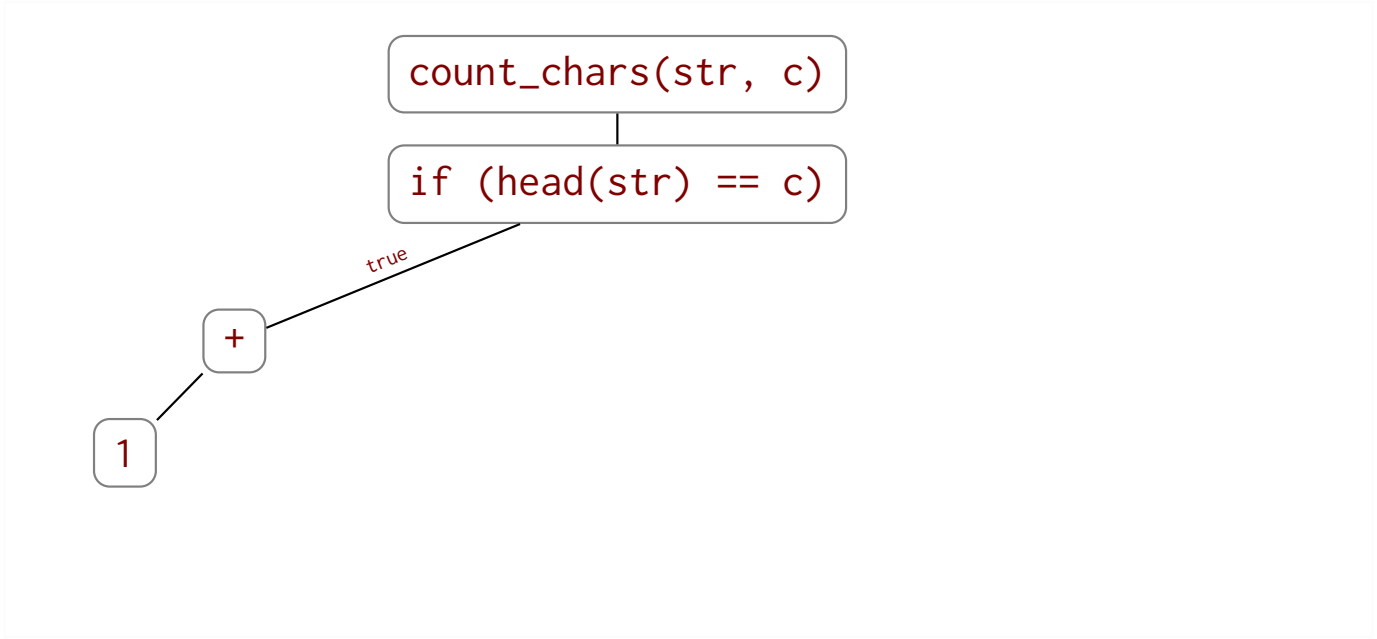


# Implémentation sous la forme d'une expression (2)

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` privé de son 1er caractère.

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

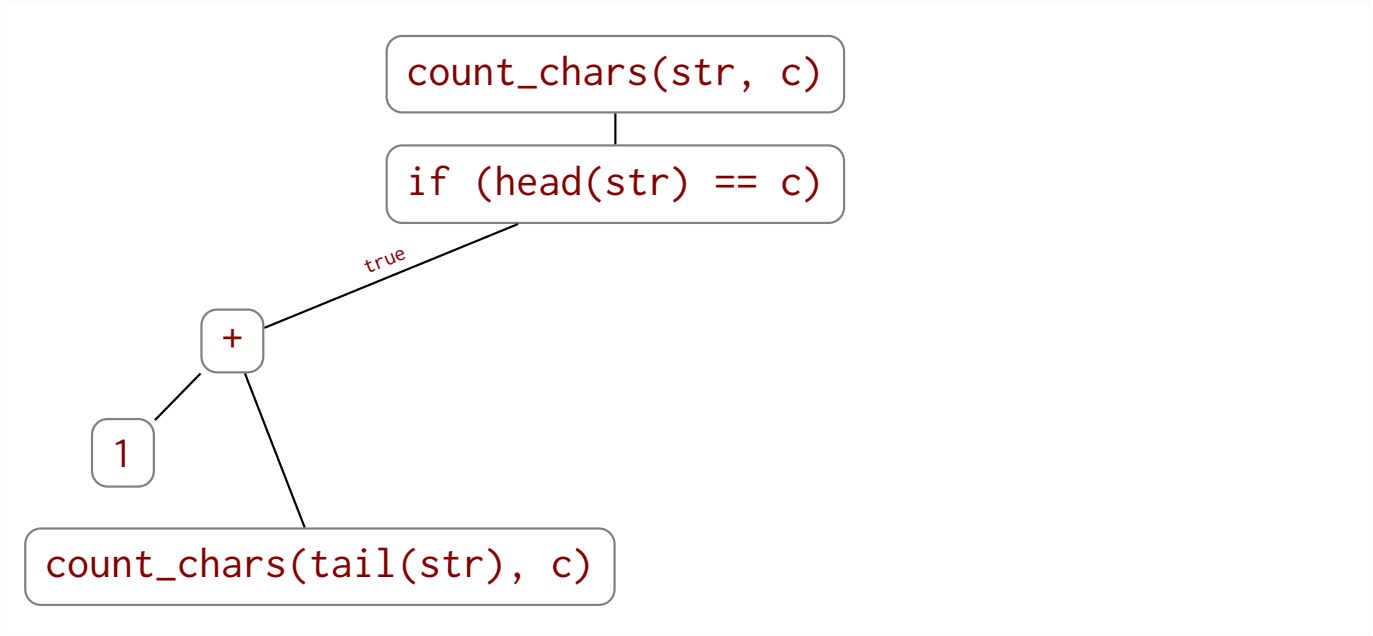


# Implémentation sous la forme d'une expression (2)

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` privé de son 1er caractère.

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

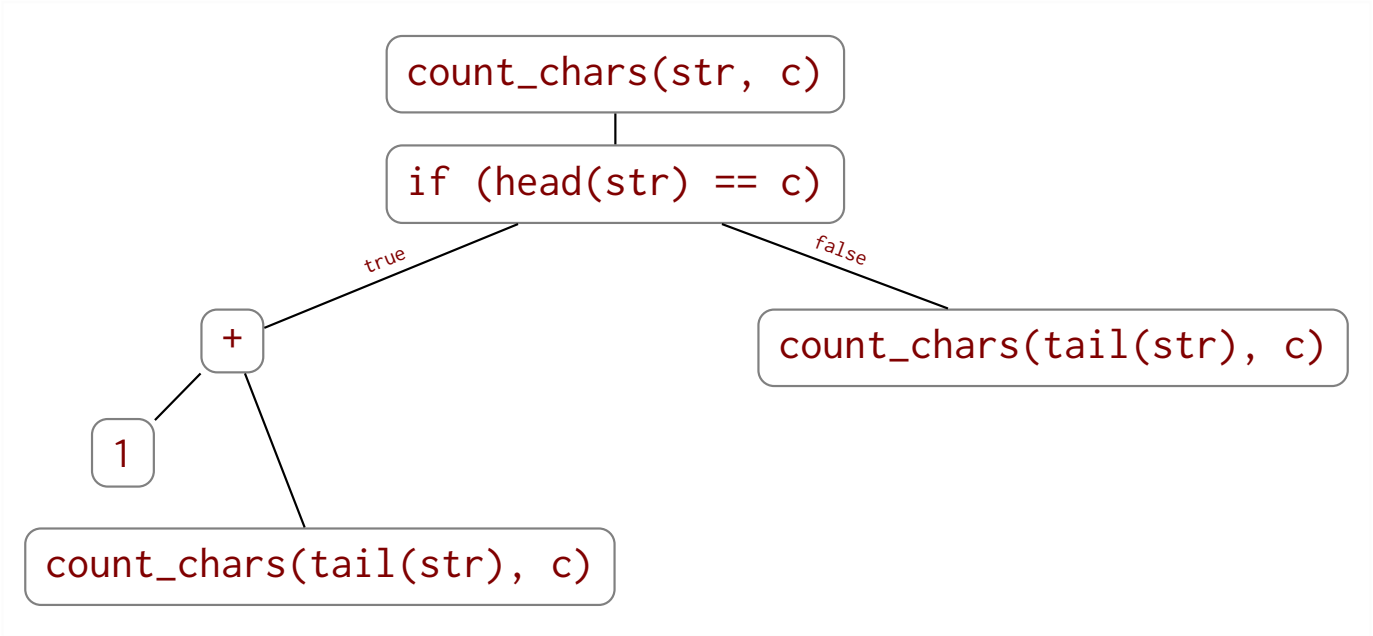


# Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

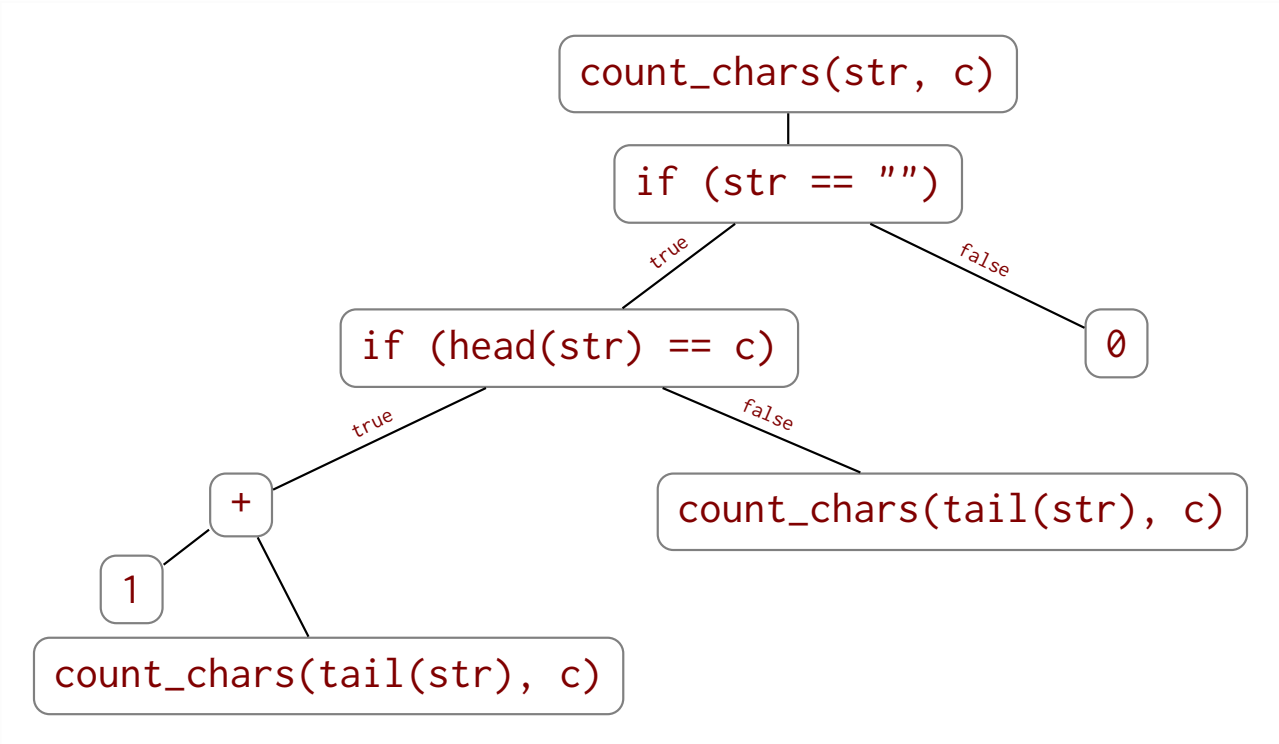
- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privé de son 1er caractère.

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**.

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

```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 de programmes comme des ensembles d'équations, privilégie les relations logiques entre objets plutôt que les représentations 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) :

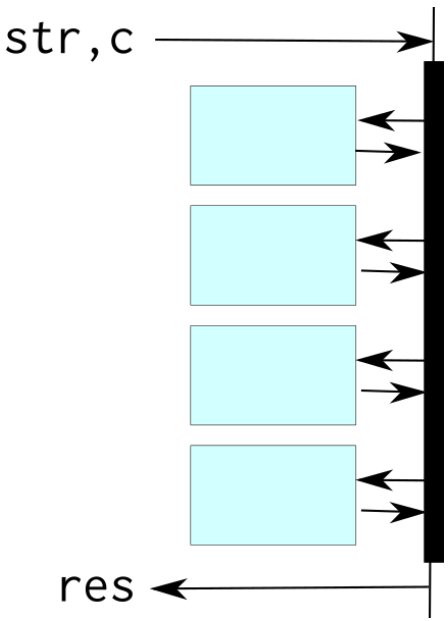
```
function countChars(str, c) {
  if (str.length == 0)
    return 0;
  else if (head(str) == c)
    return 1 + countChars(tail(str), c);
  else
    return countChars(tail(str), c);
}
```

Comparaison entre les versions :

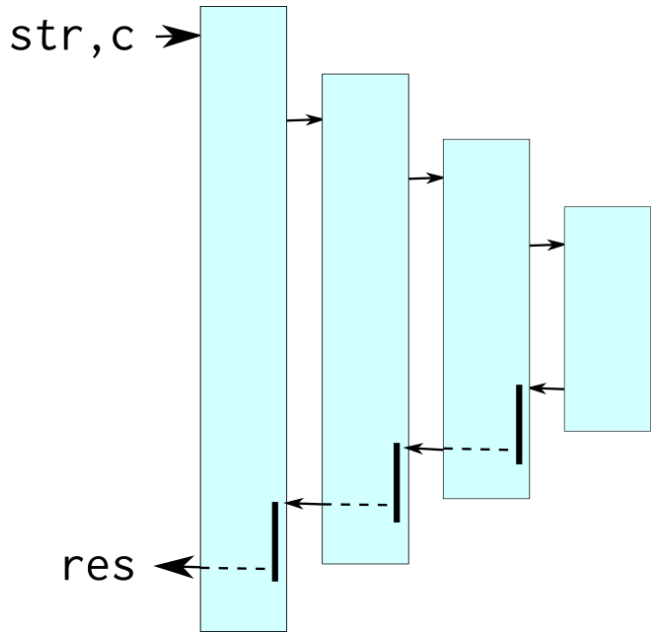
- **Abstraction** : la version (I) requiert une connaissance concrète de l'implémentation des chaînes de caractères, et de la façon d'itérer dessus ;
- **Indépendance** : les calculs de la version (F) n'interfèrent pas entre eux ; il n'y a pas d'état intermédiaire, explicite ou implicite.

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 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 ...

# 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 des calculs.

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

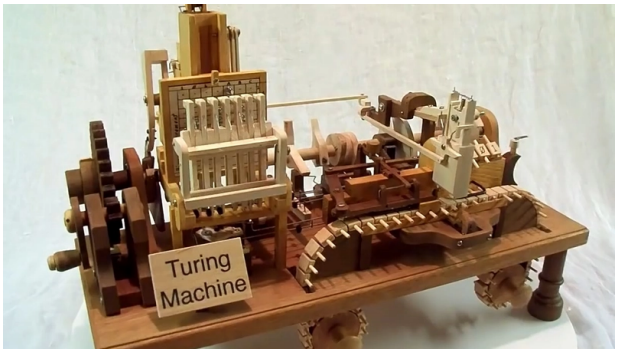
## Le lambda-calcul

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

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

- 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

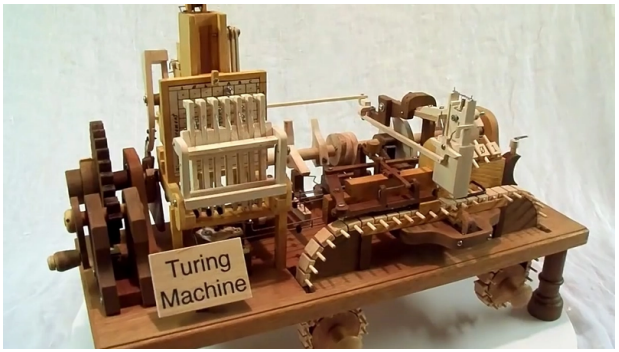
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

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

- 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

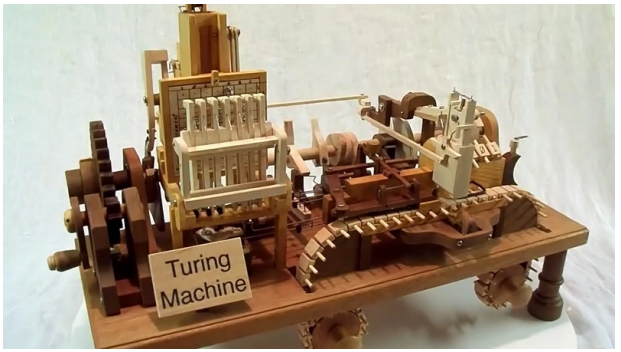
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

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

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

"Style" fonctionnel

## Les machines de Turing




Source : https://hackaday.com

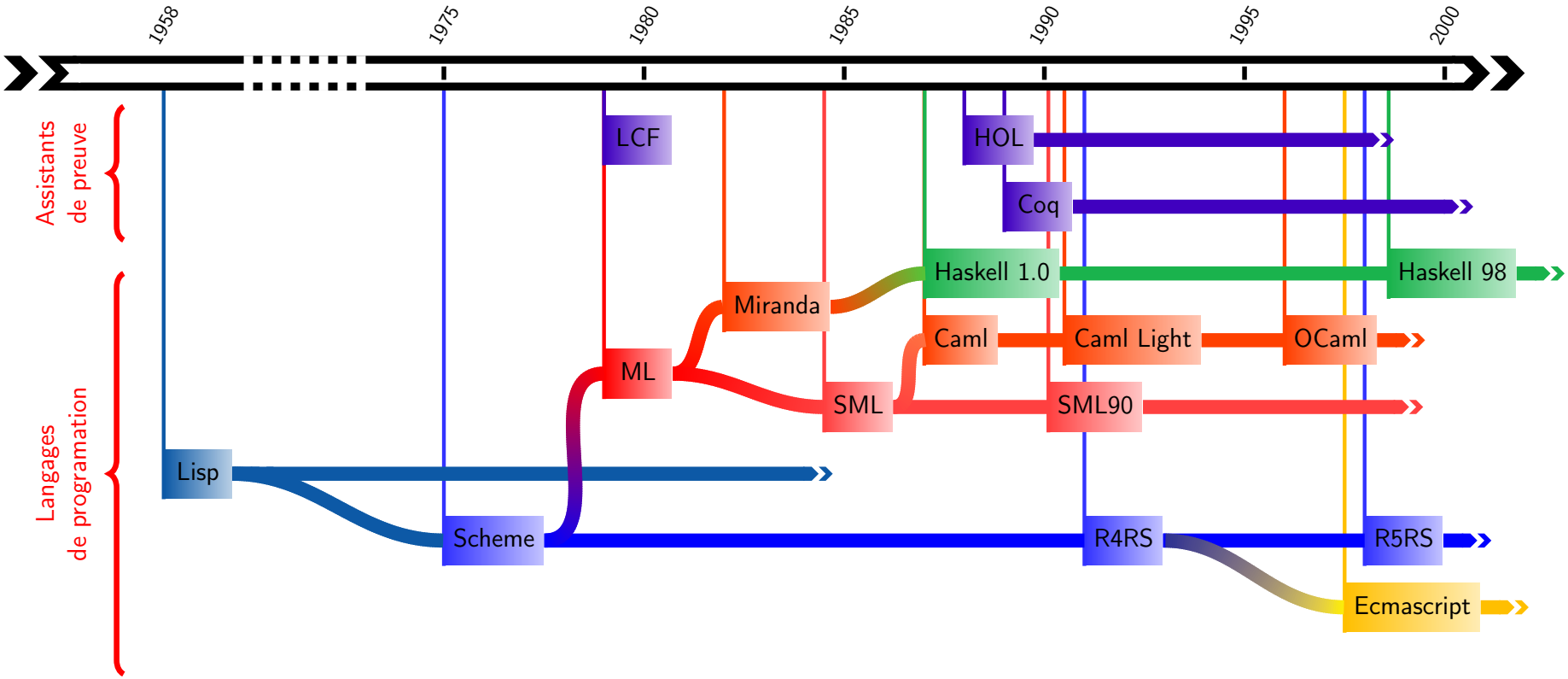
- Une machine formelle "concrète"
- Un état global modifié par des instructions

"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.

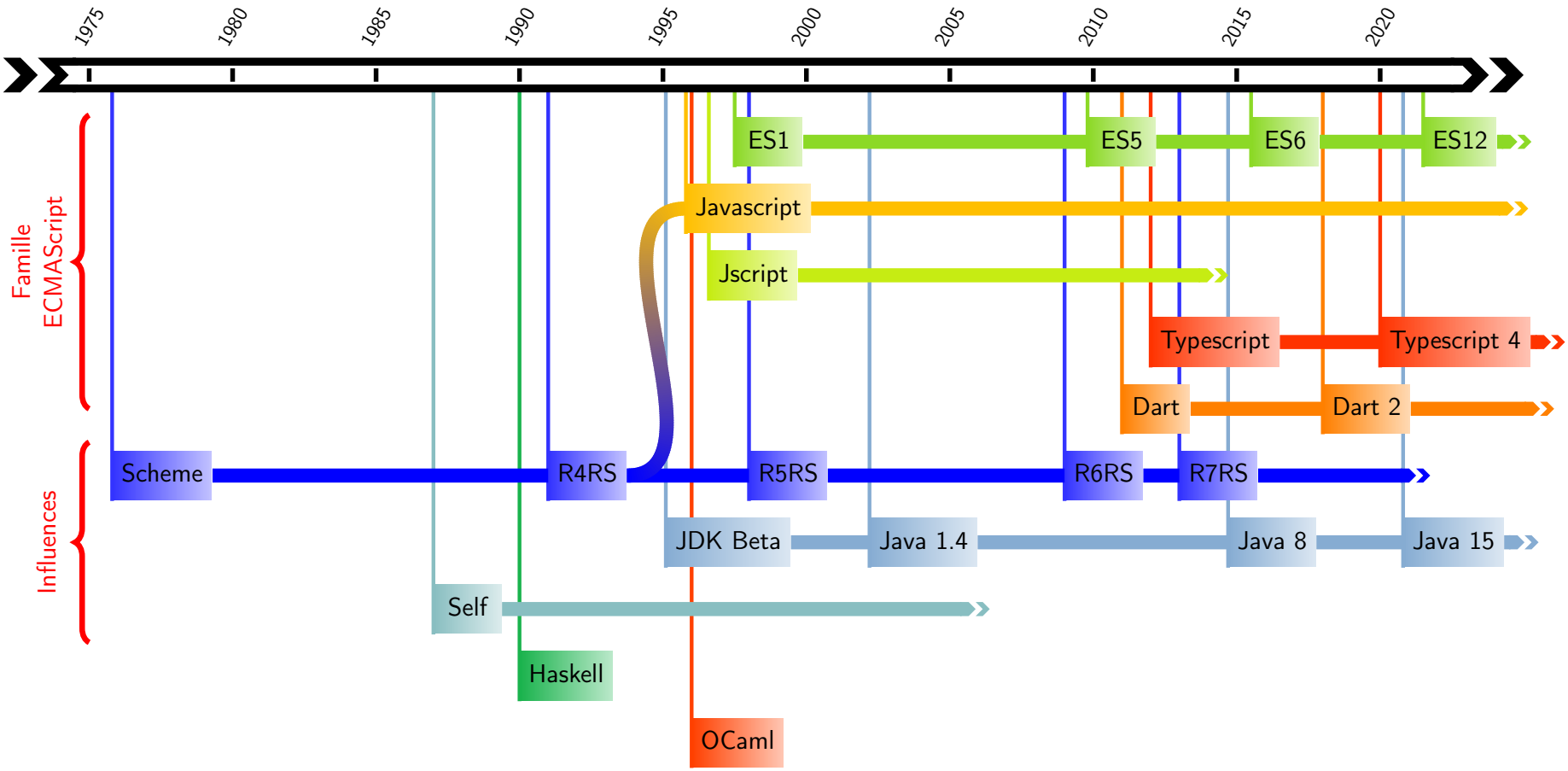
# Une brève histoire des langages fonctionnels



# 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) est 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'.
```

# 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 because it is immutable)

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

# 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);
```

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

# 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**

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

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

La variable stockée dans `aSet` est :

- allouée dans `setCreate`,
- est transmise et mise à jour lors des appels à `setAdd` et `setLength`,
- 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;
};
```

⇒

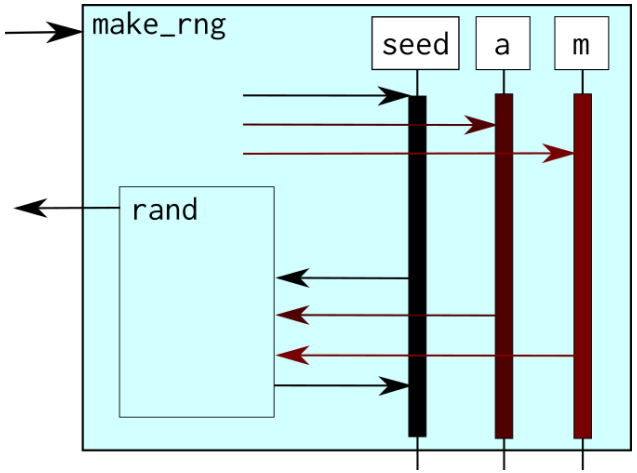
```
function makeRng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };
  return rand;
}
let rand = makeRng();
```

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.



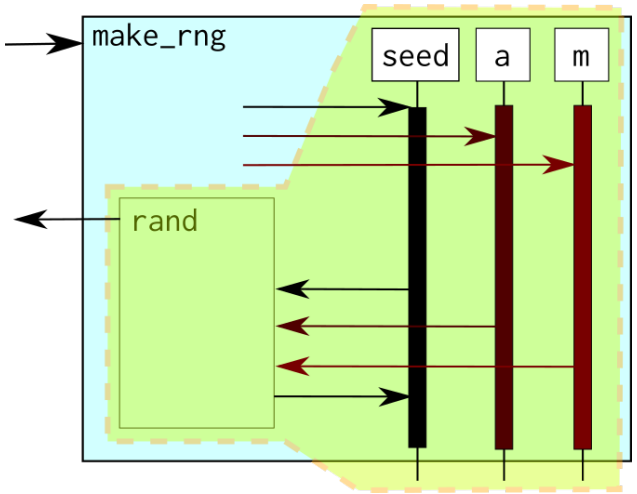

- Grâce aux fermetures, les fonctions deviennent des entités autonomes.
- Grâce aux portées, les fonctions ont une séparation public/privé.

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.

# 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 les 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 d'autres éléments externes, et produisant un élément de $E$.

- `Math.random`

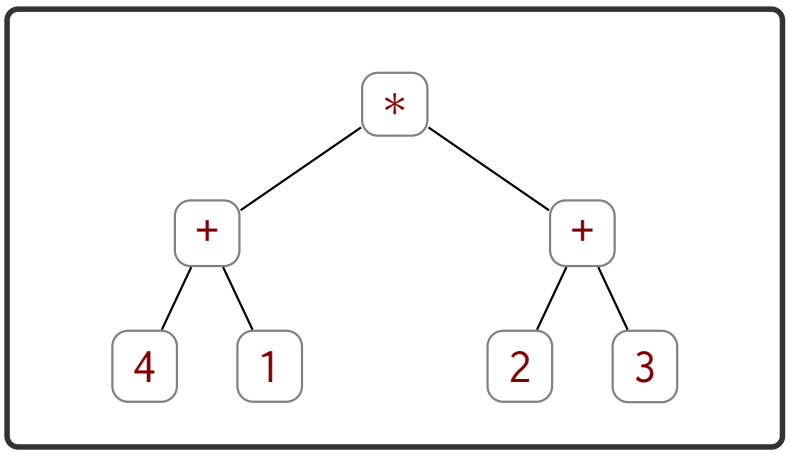
```
Math.random(); // → a number between 0 and 1
```

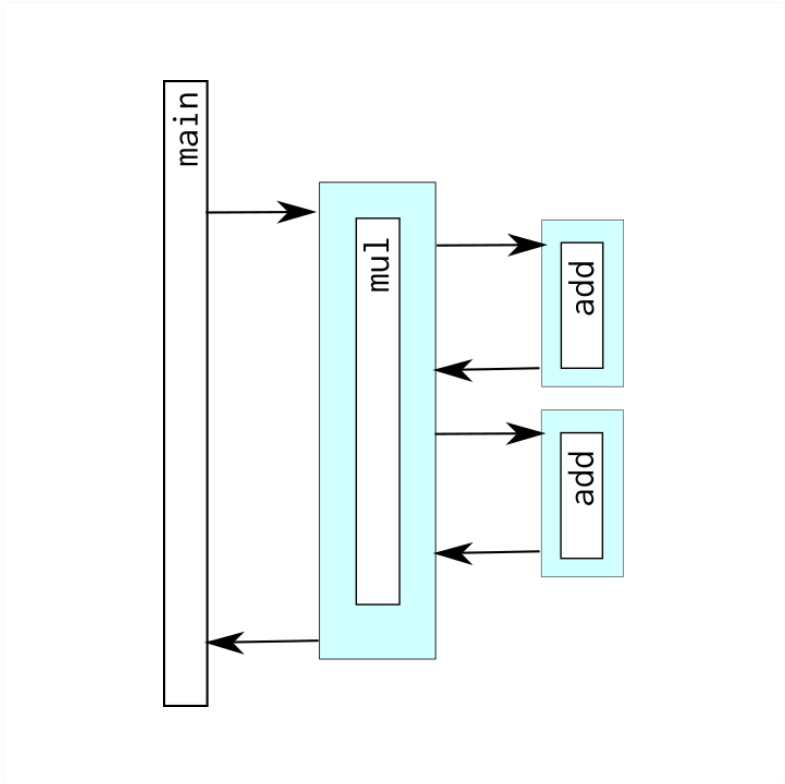
- `Date.now`

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

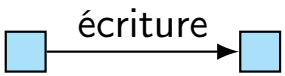
Cette différence de philosophie se traduit donc dans les programmes.

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



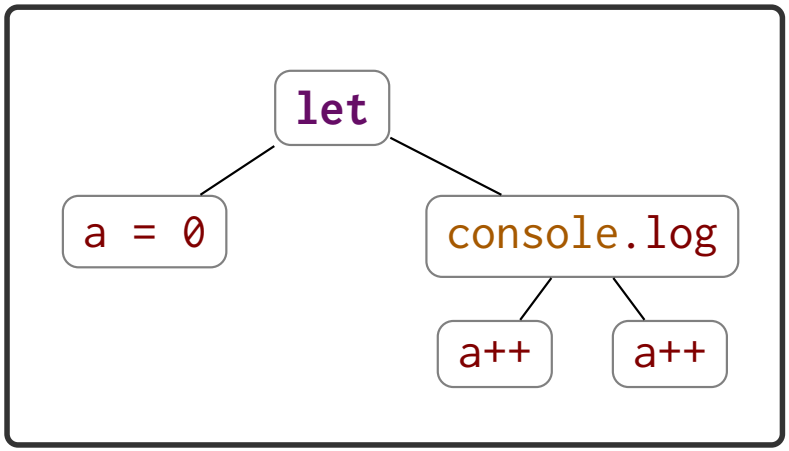



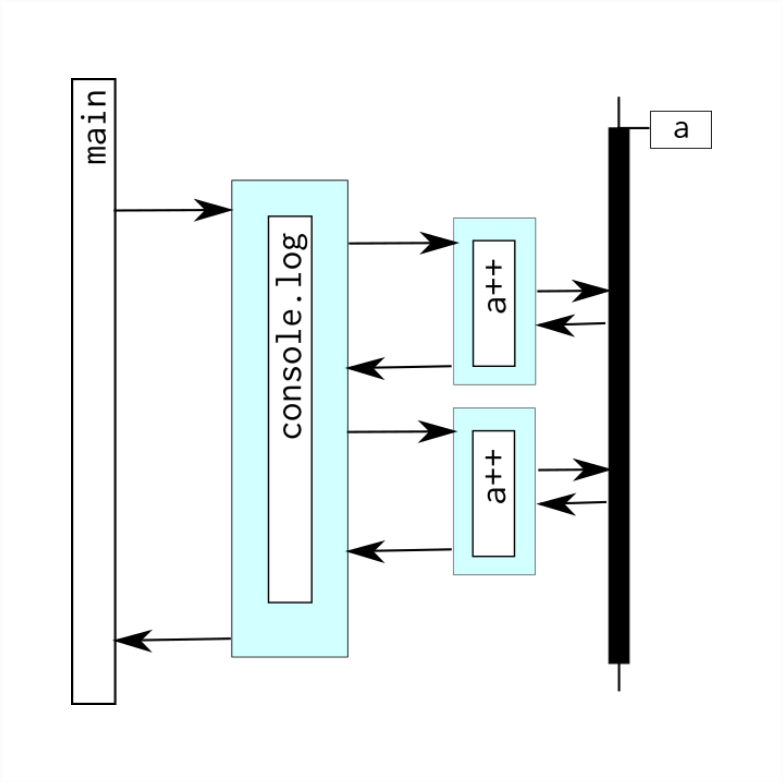

Les flèches indiquent les **dépendances** que les calculs entretiennent entre eux.



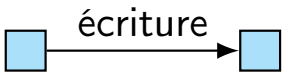

Cette différence de philosophie se traduit donc dans les programmes.

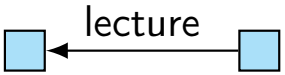
```
function main() {
  let a = 0;
  console.log(a++, a++);
}
```




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 du code plusieurs fois en séquence ou en parallèle 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 programmation**

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]]);
}
```

```
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

Application directe du remplacement appel de fonction / évaluation

```
lemma appendLengthOk(l1 : List, l2 : List)
ensures length(l1) + length(l2) == length(append(l1, l2)) { // ← statement of the lemma
    match (l1)
    case Nil ⇒ calc {              length(append(l1, l2));
                                   ==
                                   length(l2); }                        // def. append

    case Cons(hd1, tl1) ⇒ calc {   length(append(l1, l2));
                                   ==
                                   length(Cons(hd1, append(tl1, l2))); // def. append
                                   ==
                                   1 + length(append(tl1, l2));        // def. length
                                   == { appendLengthOk(tl1, l2); }     // induction step
                                   1 + length(tl1) + length(l2);
                                   ==
                                   length(l1) + length(l2); } }        // def. 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 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**.

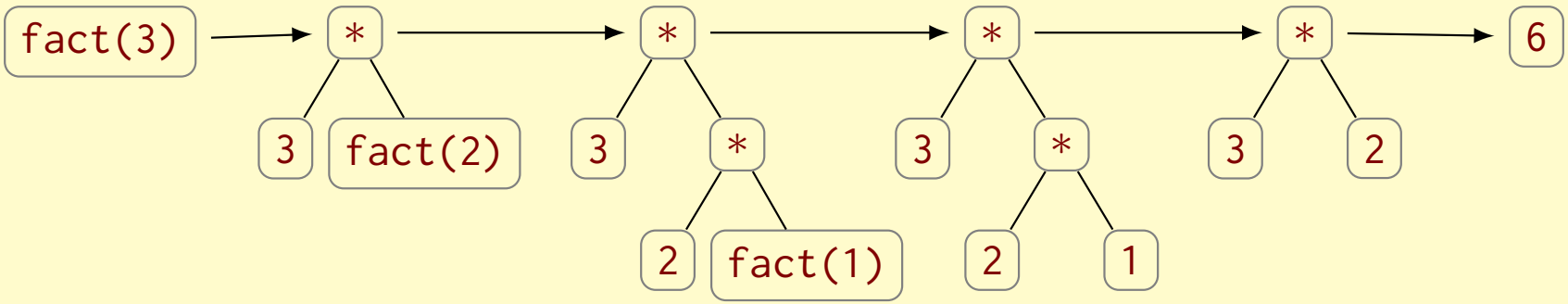
## Définition (Récursif)

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.

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



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 ...)

Cette forme correspond à une décomposition formelle connue, facilitant :

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

Mais il s'agit aussi d'une forme permettant des optimisations :

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

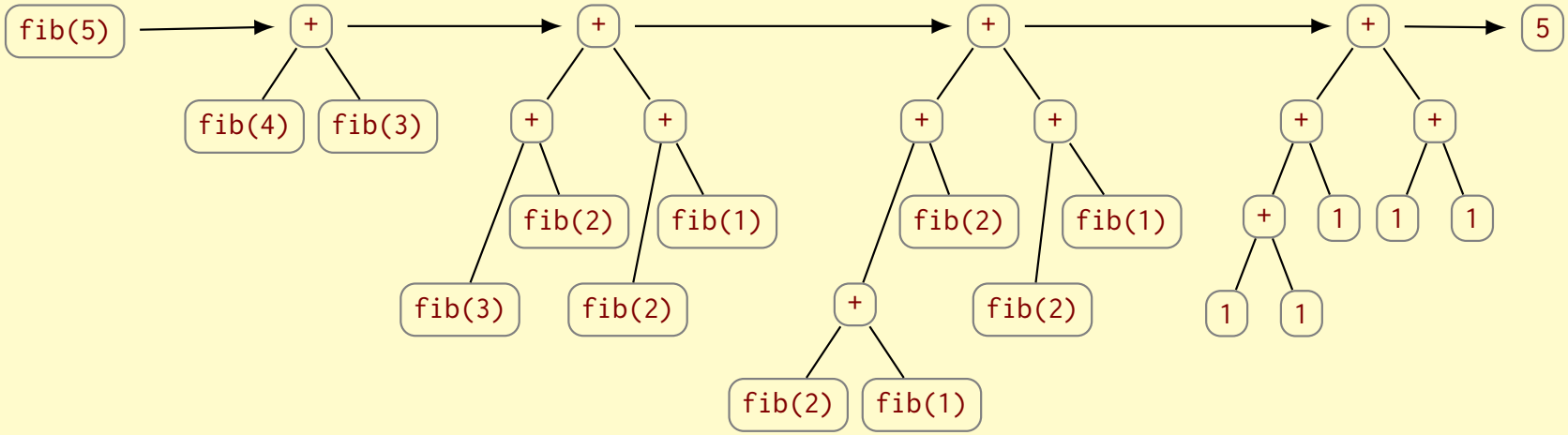
## Limites des algorithmes récursifs

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” :

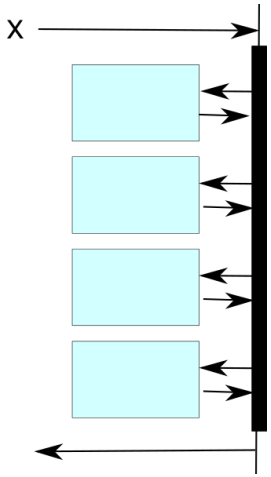
- La boucle classique :

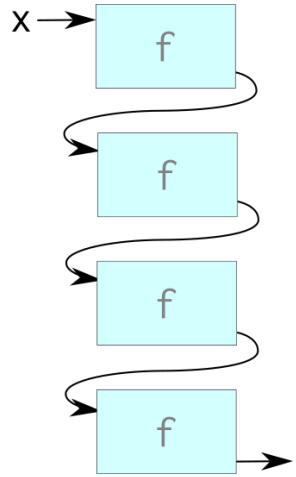
```
function loop(block, init, times) {
  let res = init;
  for (let i = 0; i < times; i++) {
    res = block(res);
  }
  return res;
}
```

- 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 finalement très proches.
- Ils n'ont pas les mêmes qualités en terme de pureté.
- La version récursive "simple" n'est pas universelle :
  elle n'est pas représentative de tous les algorithmes récursifs,
  (mais toute boucle peut se mettre sous cette forme)

# 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

## Fait

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 le demande, les machines Javascript actuelles n'optimisent pas les appels récursifs terminaux.

# Optimisation des appels récursifs terminaux

## Fait

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 le demande, les machines Javascript actuelles n'optimisent pas 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

Mise sous forme récursive terminale 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

Mise sous forme récursive terminale 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

Mise sous forme récursive terminale 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);
}
```

# 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) {
     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.

# Types 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 (Type de données fonctionnel)

Un type de données $\mathcal{T}$ est dit (purement) **fonctionnel** si il peut être implémenté de manière pure. Un tel type est forcément immutable.

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.

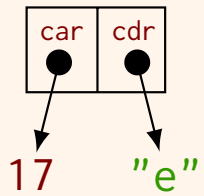
- Nommés aussi types de données algébriques, types sommes ou variants.

```
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** (*cons*) est une structure de donnée contenant deux références appelées traditionnellement `car` et `cdr`.



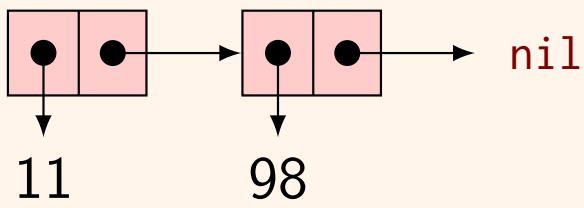

```
{ car: 17, cdr: "e" }
```

```
// constructor
function cons(_car, _cdr) { return { car: _car, cdr: _cdr }; }
// accessors
function car(cons)        { return cons['car']; }
function cdr(cons)        { return cons['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 différente (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.

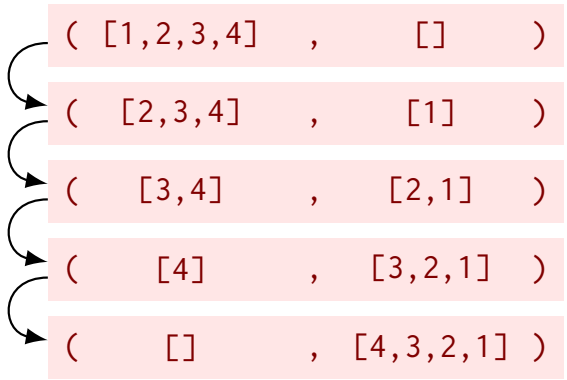
**Influence de la structure inductive**

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

# Exemple d'algorithme sur les listes : reverse

Considérons le problème consistant à retourner 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`).

```
const L = require("list");
function countChars(str, c) {
    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
}
```

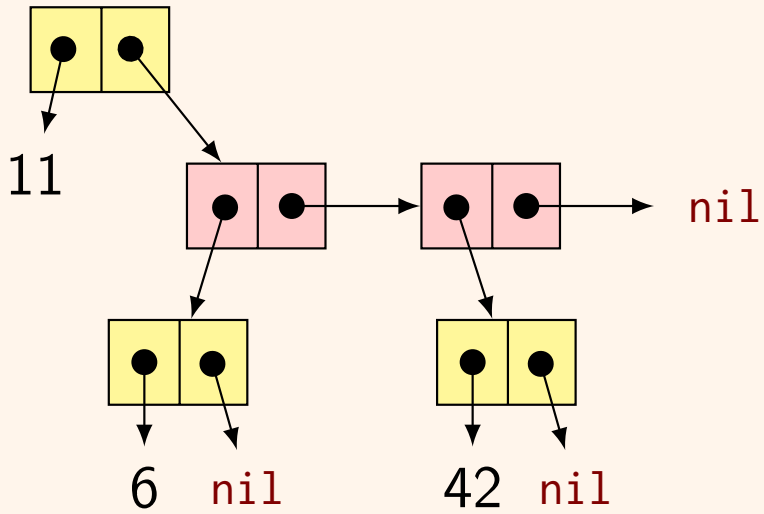
Parmi les caractéristiques mises en avant :

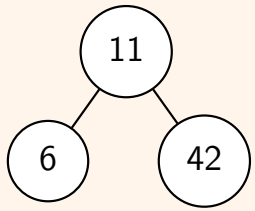
- les listes sont immutables, et donc adaptées à la programmation pure ;
- les listes sont immutables, permettant des optimisations comparables à ce que l'on peut obtenir avec des tableaux.

## 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(_val, _chld) { return { val: _val, children: _chld }; }
function val(tree)         { return tree['val']; }
function children(tree)    { return tree['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 $\Rightarrow$ nécessité d'apporter d'autres fonctions.

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.

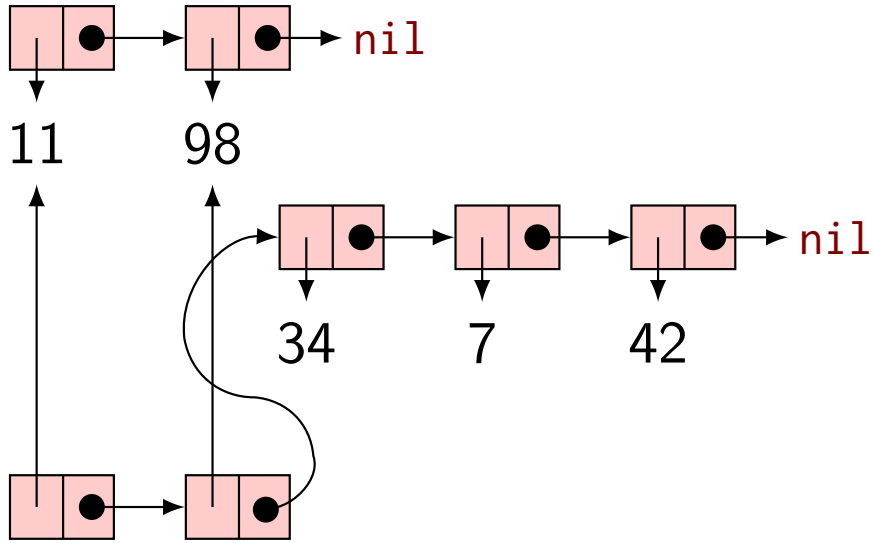
**Fait**

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 (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

Le coût de la persistance peut être contenu dans des limites raisonnables.

# 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 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;
}
```

$$\texttt{loop} : (\texttt{block} \times \texttt{init} \times \texttt{times}) \rightarrow \texttt{res}$$

$$\texttt{block} : \texttt{in} \rightarrow \texttt{out}$$

# 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;
}
```

$$\underset{\quad}{\texttt{loop}} : \underset{\texttt{T}}{\texttt{block}} \times \underset{\texttt{number}}{\texttt{init}} \times \underset{\texttt{T}}{\texttt{times}} \rightarrow \underset{\texttt{T}}{\texttt{res}}$$

$$\texttt{block} : \underset{\texttt{T}}{\texttt{in}} \rightarrow \underset{\texttt{T}}{\texttt{out}}$$

- 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**.

# Ordre supérieur : l'itérateur forEach

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

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

# 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]
```

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

# Ordre supérieur : la sélection filter

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

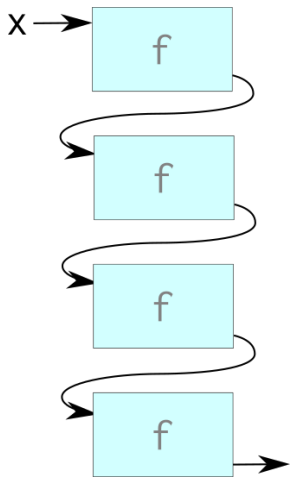
```
function filter<T>(block: (T) ⇒ boolean, arr: T[]) : T[] {
    const brr = [];
    for (let i = 0, j = 0; i < arr.length; i++) {
        if (block(arr[i]))
            brr[j++] = arr[i];
    }
    return brr;
}

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

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

# Ordre supérieur : les pliages (1/3)

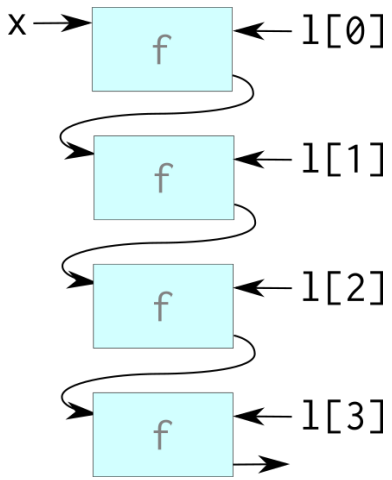
Quelle est la version fonctionnelle pure d'un parcours sur une liste ?
Pour rappel, nous avions établi le schéma suivant pour une simple boucle :




```
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 ?
Une idée simple consiste à passer les éléments de la liste 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 :


`0`

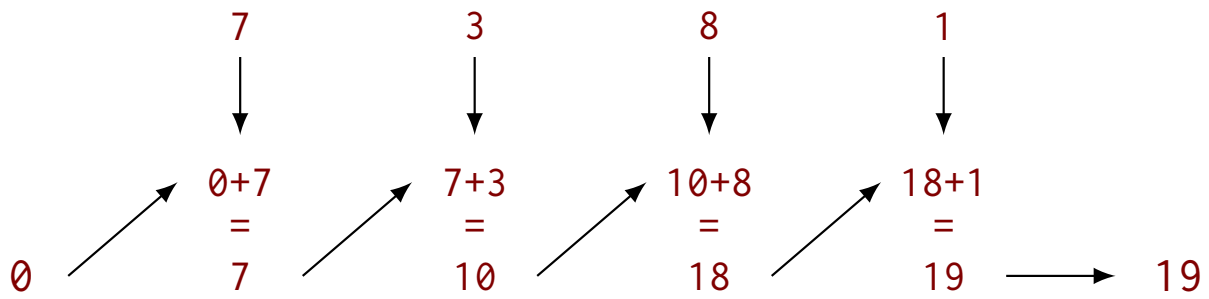
# Ordre supérieur : les pliages (2/3)

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 (3/3)

Quelques exemples d'application de `_.reduce` de la bibliothèque `underscore` :
(exemples réalisés avec `_ = require("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 somme

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

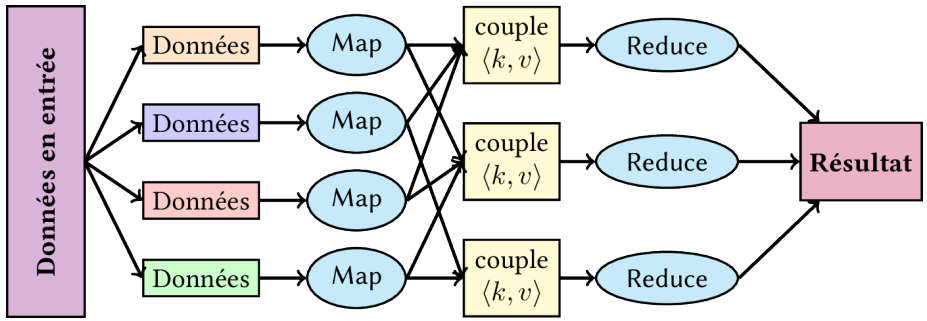
```
function treeSizeHigher(t) {
   const childrenSizes = map(treeSizeHigher, children(t));
   const sumSizes      = reduce((acc, el) ⇒ acc+el, childrenSizes, 0);
   return 1 + sumSizes;
}
```

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

# Application : le framework map-reduce

## Idée

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



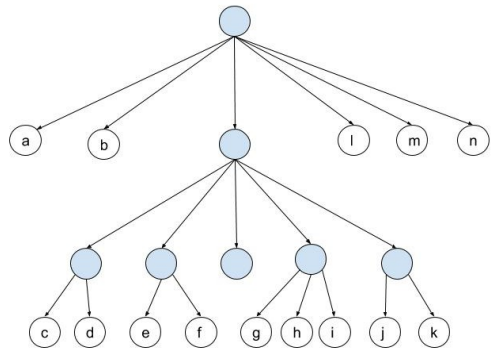

Source : Wikipedia

- 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` orientée sur les types 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.

# 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.


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