PG104 - Programmation Fonctionnelle

D. Renault

ENSEIRB-Matmeca

20 janvier 2025, v.1.2.5

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

Introduction

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

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

Ce que ce cours n'est pas

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

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

Le choix de Javascript

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

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

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

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

Les grandes lignes de ce chapitre...

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

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

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

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

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

Un petit problème simple (1/2)

Problème (Trinôme)

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

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

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

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

Un petit problème simple (2/2)

Ces équations se transforment alors naturellement en algorithme :

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

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

Idée générale

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

Un problème plus complexe (1/2)

Considérons un monde constitué :

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

Le monde en question suit les règles implicites suivantes :

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

Problème (Escape Game)

Comment ouvrir la porte pour sortir de la pièce ?

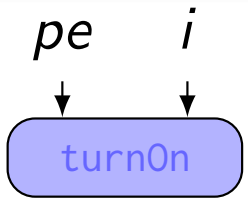
Un problème plus complexe (2/2)

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

Objectif

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

Un problème plus complexe (2/2)



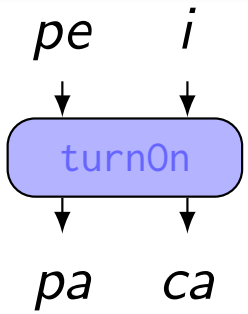
$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

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

Un problème plus complexe (2/2)

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

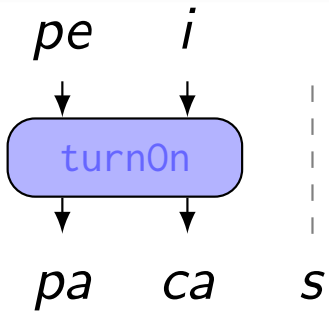


Objectif

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

Un problème plus complexe (2/2)

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

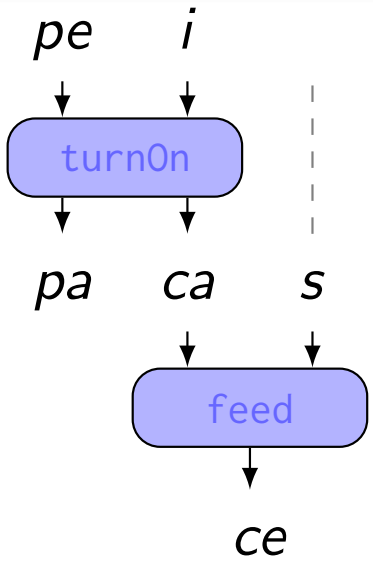


Objectif

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

Un problème plus complexe (2/2)

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

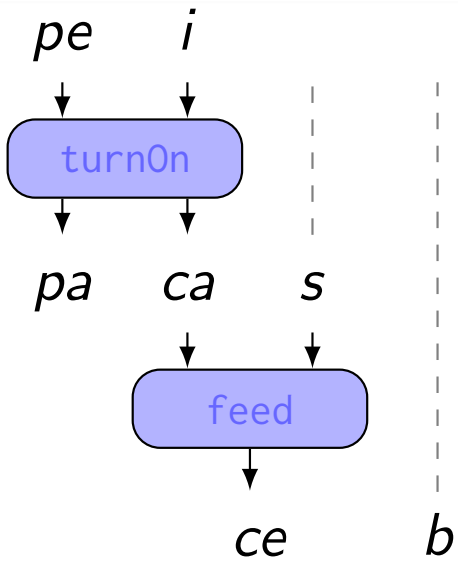


Objectif

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

Un problème plus complexe (2/2)

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



Objectif

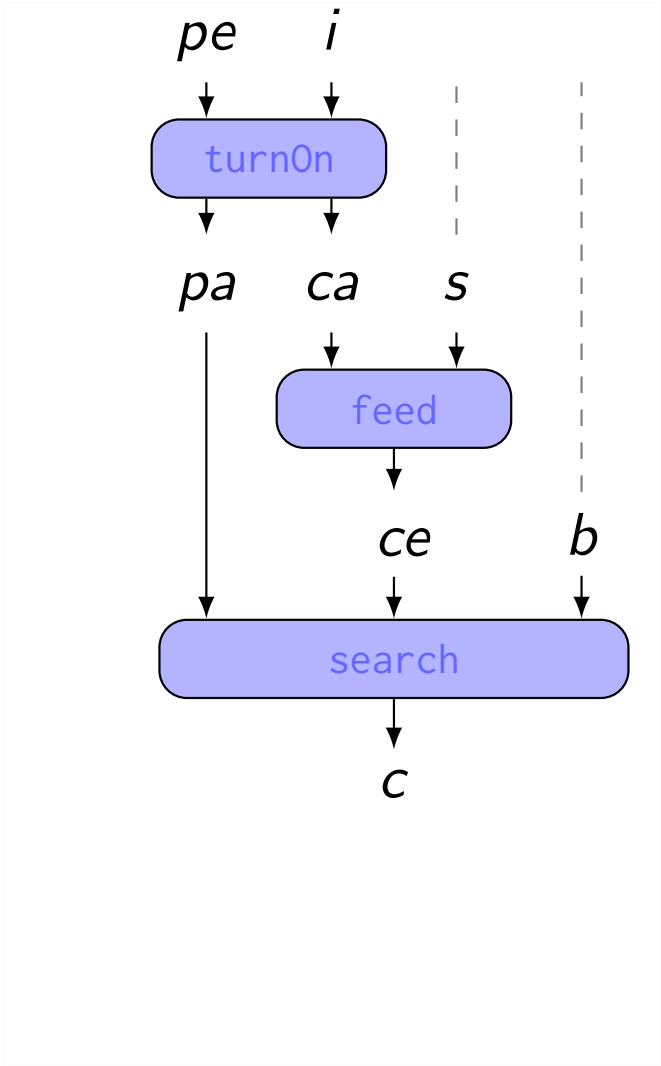
À partir de pe, pf, i, b, s, produire po.

Un problème plus complexe (2/2)

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

Objectif

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

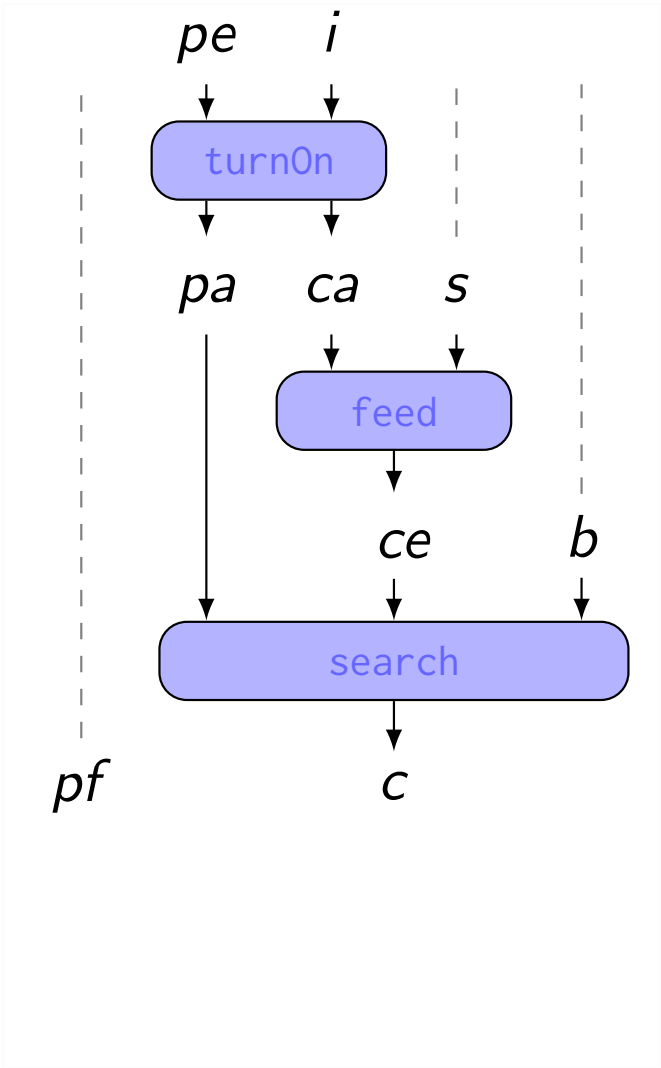


Un problème plus complexe (2/2)

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

Objectif

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

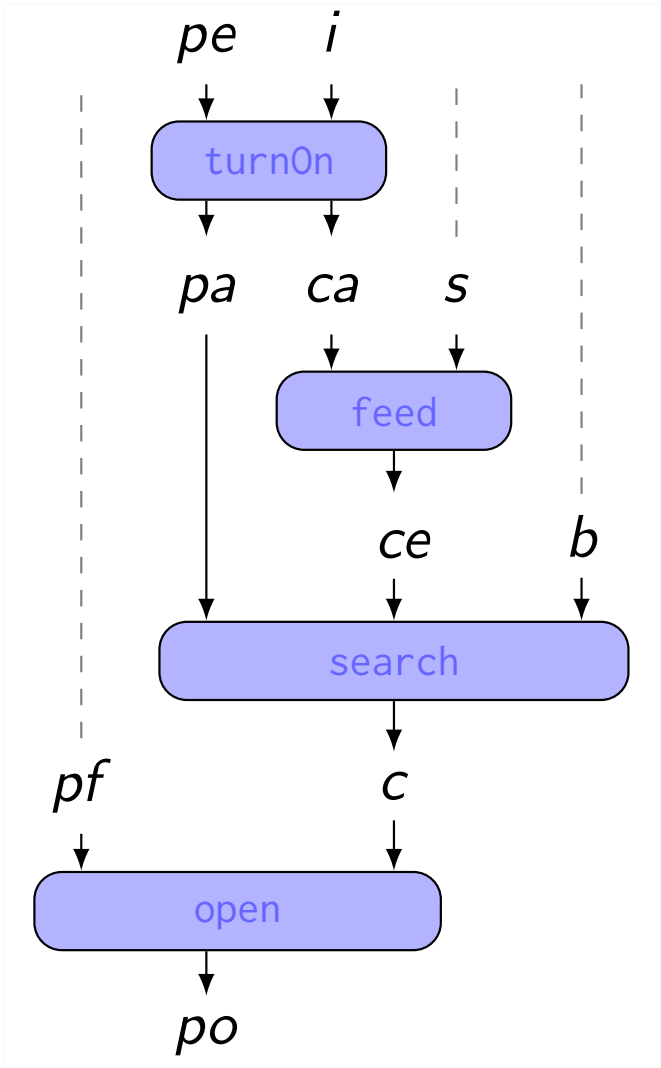


Un problème plus complexe (2/2)

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

Objectif

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



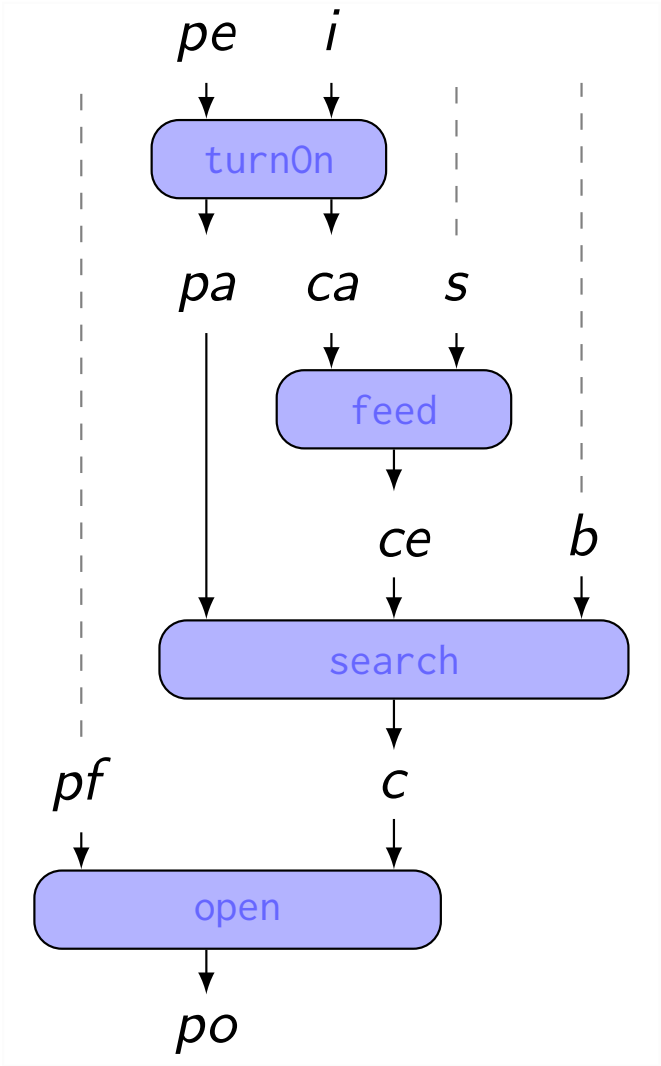
Un problème plus complexe (2/2)

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

Objectif

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

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



Examinons deux manières de modéliser une simplification du problème :

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

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

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

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

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

Examinons deux manières de modéliser une simplification du problème :

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

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

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

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

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

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

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

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

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

Examinons deux manières de modéliser une simplification du problème :

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

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

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

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

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

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

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

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

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

Différentes manières qui induisent différentes qualités du code final :

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

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

Paradigmes de programmmation

Définition (Paradigme [Nar05])

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

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

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

Des idées possibles de paradigmes

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

Programming paradigms

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

Source : Wikipedia

Le paradigme impératif

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

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

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

Le paradigme fonctionnel

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

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

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

Expression

Considérons un langage de programmation contenant :

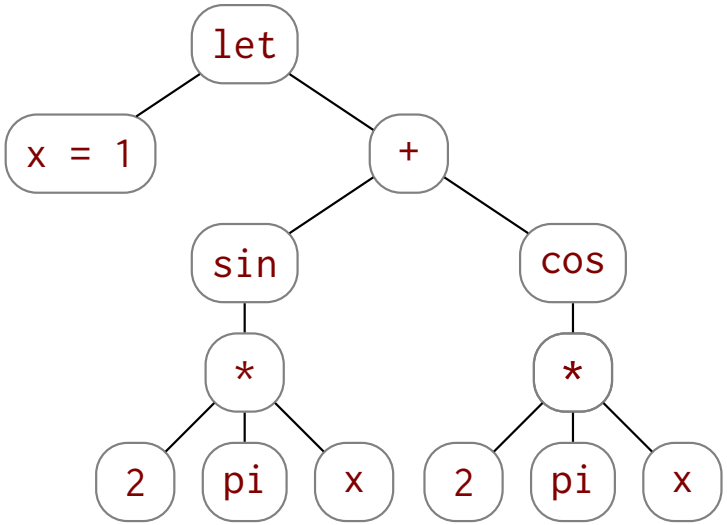
- un ensemble de **valeurs** $\mathcal{V}$ (entiers, chaînes de caractères, booléens ...) ;
- un ensemble $\mathcal{T}$ de **moyens de composer** ces valeurs (fonctions, méthodes, constructions syntaxiques particulières ...).

Définition (Expression)

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

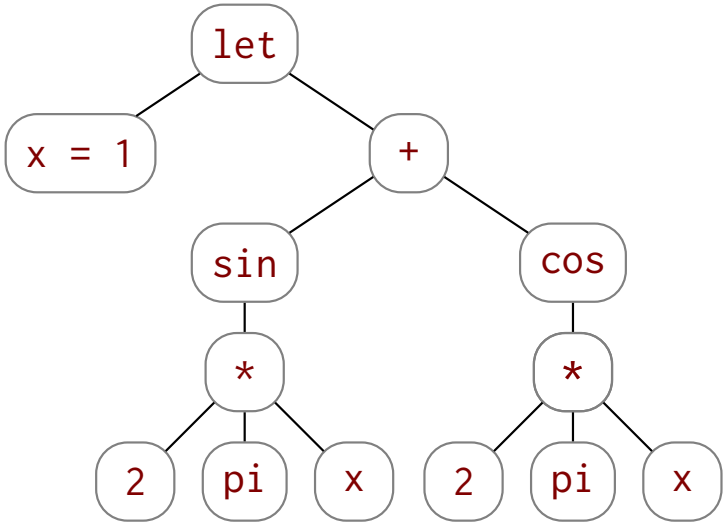
- L'expression est la représentation arborescente d'un calcul.
- **Évaluer** une expression permet d'obtenir le résultat du calcul.

Exemples d'expressions (1/2)




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

Exemples d'expressions (1/2)



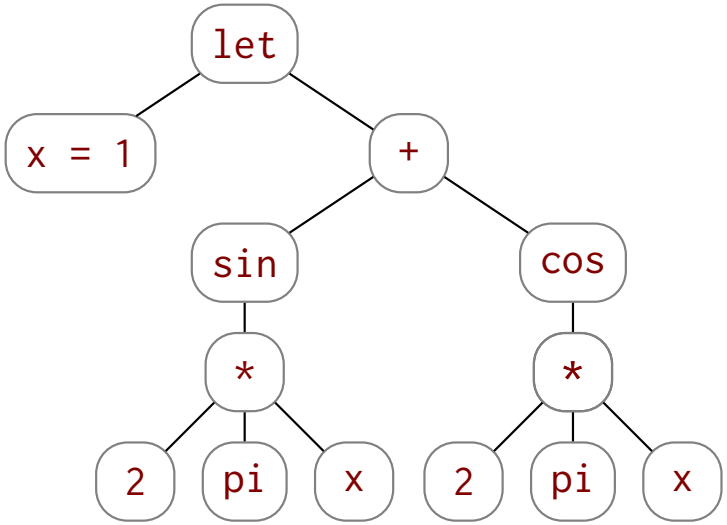

```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Haskell) :

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

-- → 0.9999999999999998
```

Exemples d'expressions (1/2)

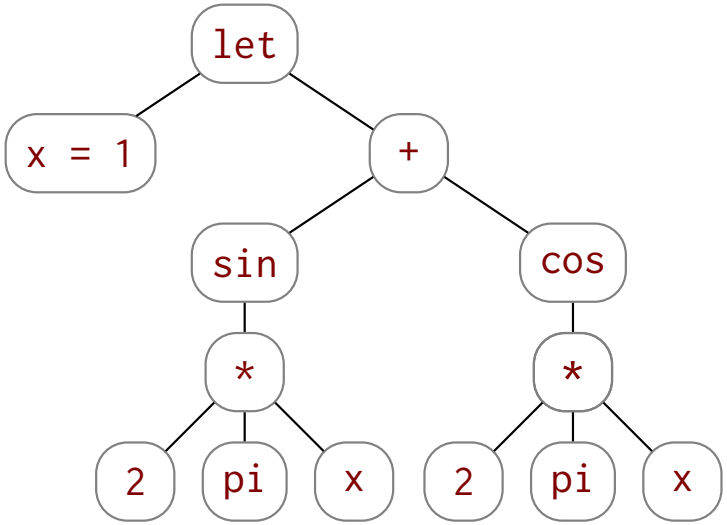



```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Javascript) :

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

Exemples d'expressions (1/2)



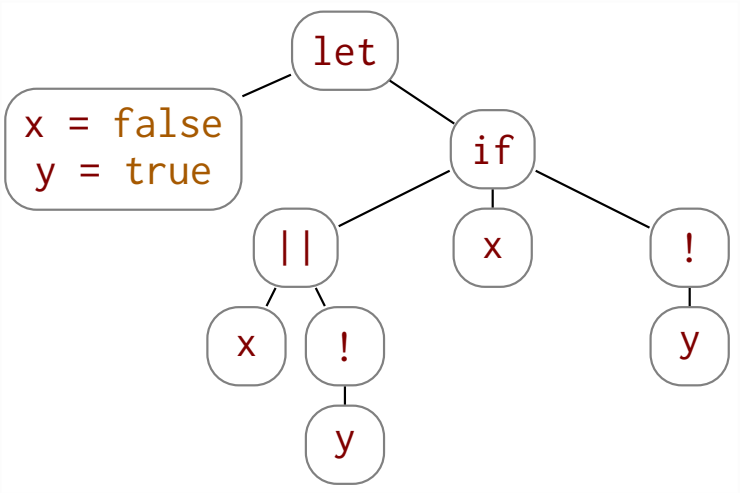

```
let x = 1 in sin(2*pi*x) + cos(2*pi*x)
```

Évaluation (Javascript) :

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

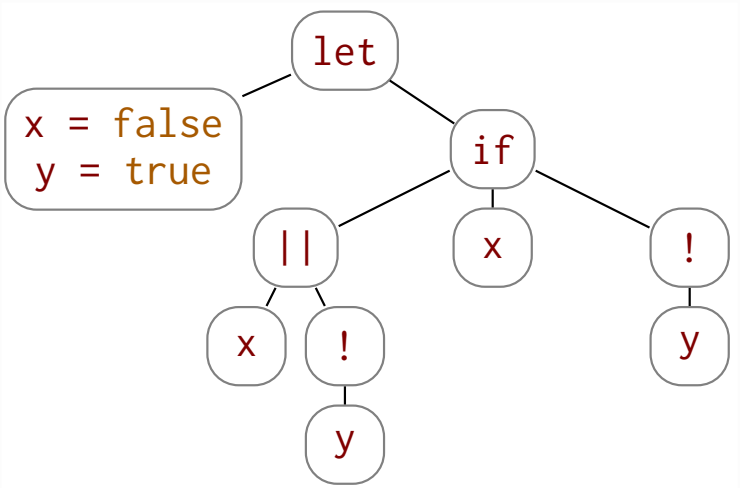
L'évaluation d'une expression se fait dans un **environnement** contenant les valeurs des objets accessibles à un moment de l'évaluation.

Exemples d'expressions (2/2)




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

Exemples d'expressions (2/2)



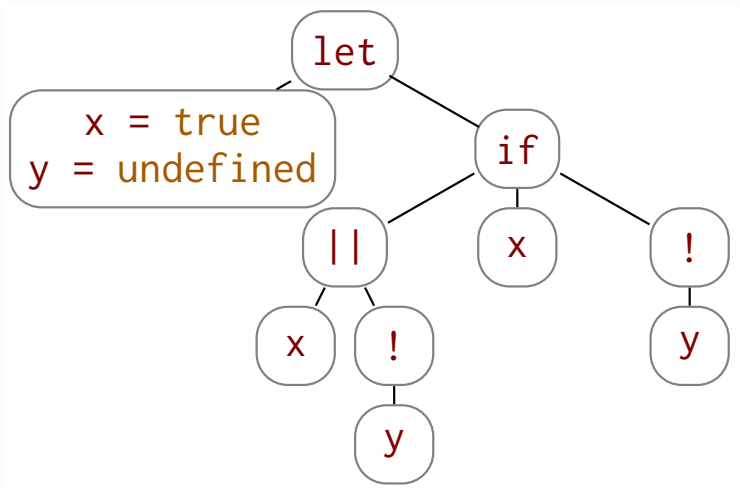

```
let x = false and y = true in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

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

Ternary conditional operator

Exemples d'expressions (2/2)

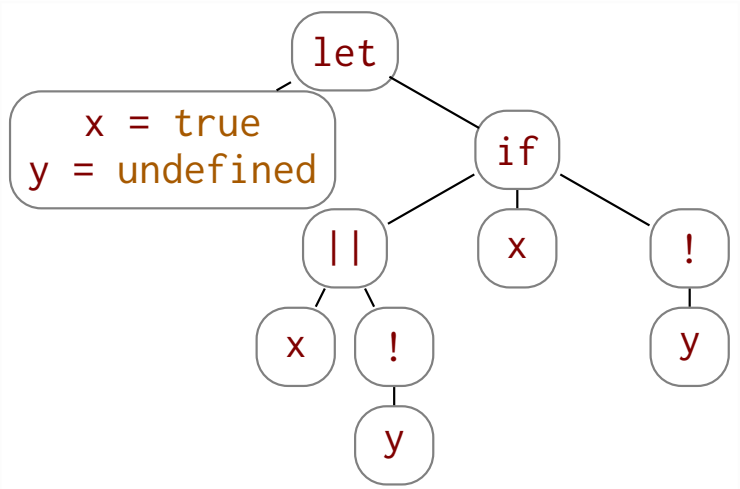



```
let x = true and y = undefined in
    if (x || !y) then x else !y
```

Évaluation (Javascript) :

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

Exemples d'expressions (2/2)




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

Évaluation (Javascript) :

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

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

Programmer uniquement avec des expressions ?

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

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

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

Exemple d'application : le comptage de caractères

Comparons les paradigmes impératif et fonctionnel sur un exemple.

Problème (Comptage de caractères)

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

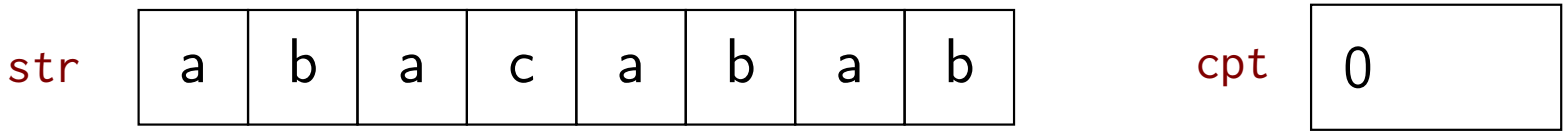
Exemples

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

Implémentation impérative

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

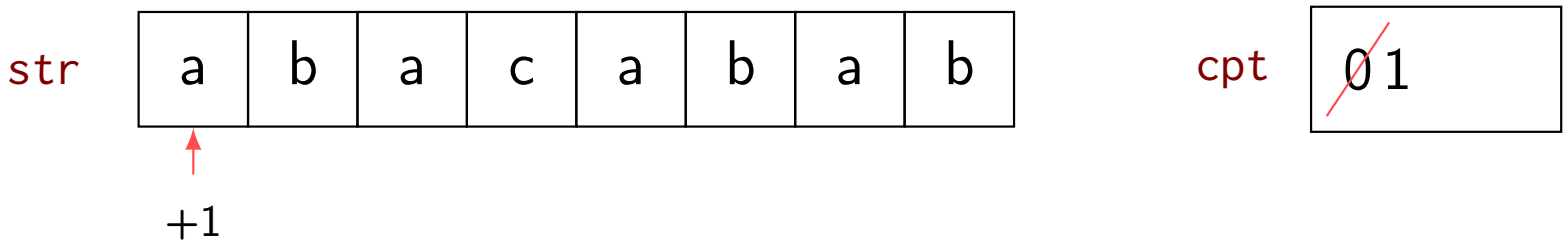
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

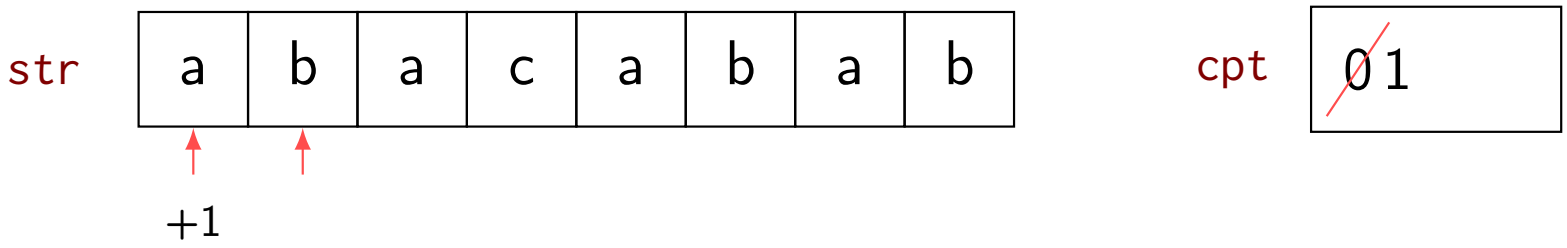
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

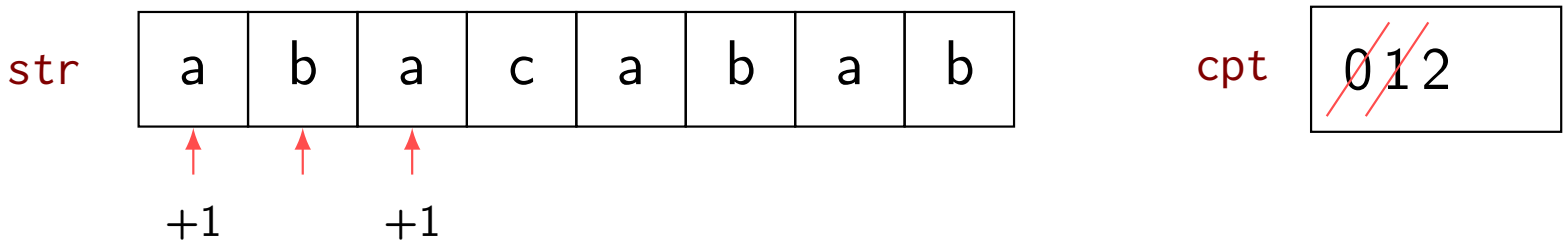
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

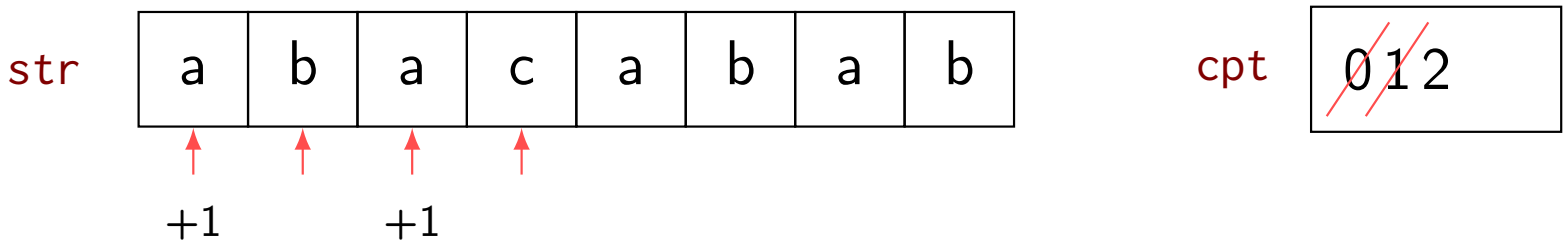
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

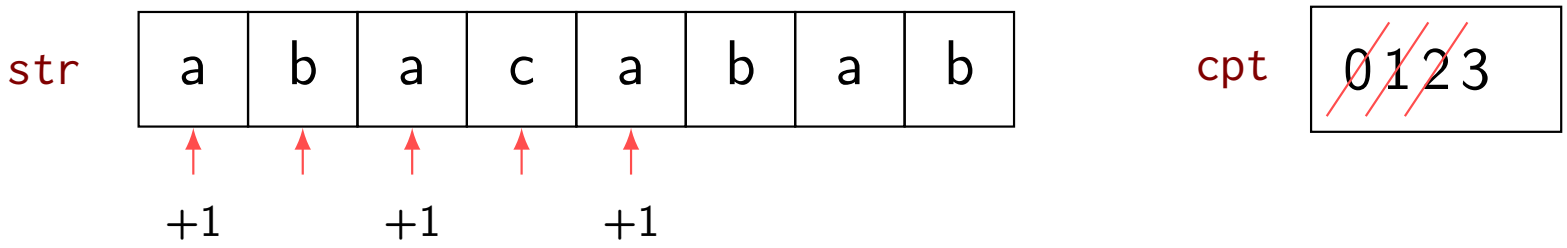
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

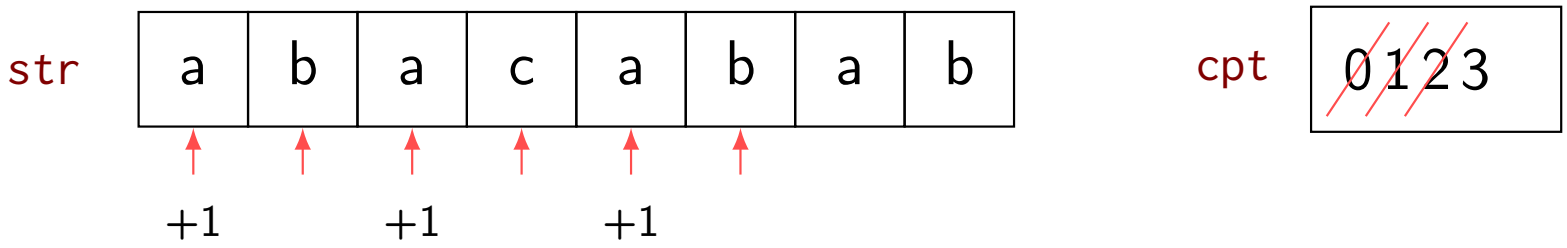
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

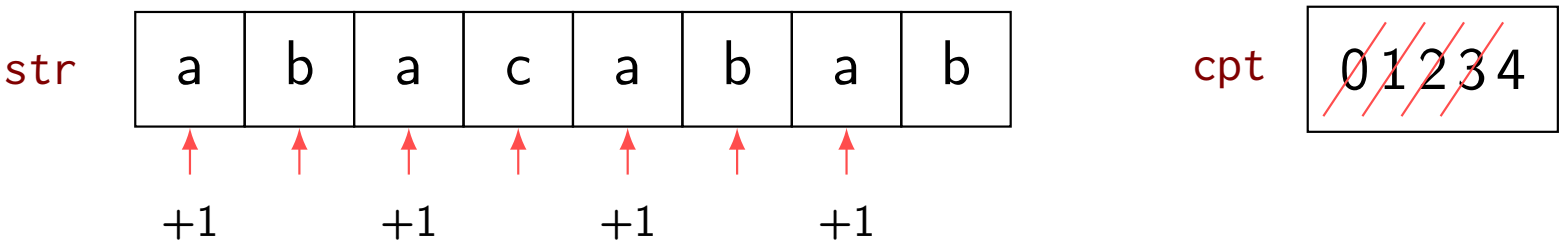
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

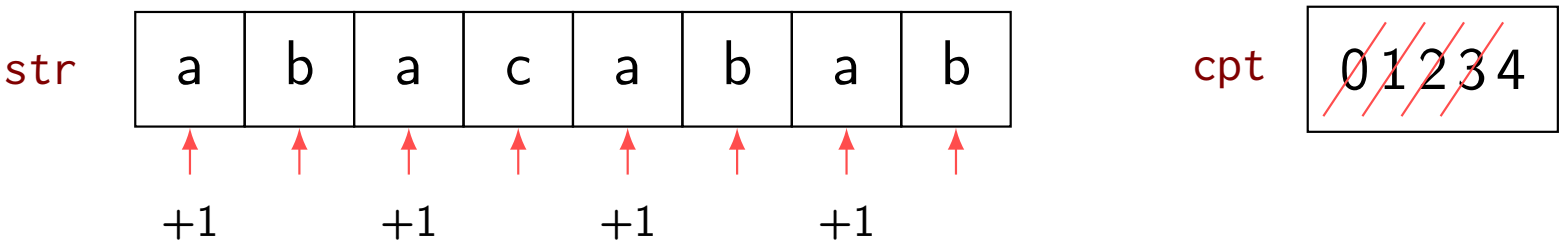
- Présence d'un compteur, d'un curseur, et d'une variable résultat.
- Suite d'instructions construisant le résultat final dans `cpt`.



Implémentation impérative

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

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



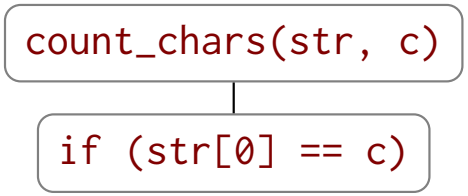
Implémentation sous la forme d'une expression (1)

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

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

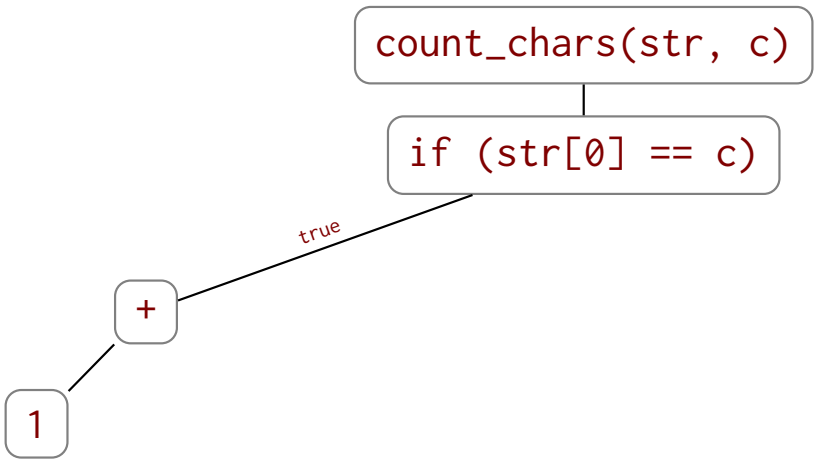
Implémentation sous la forme d'une expression (1)

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



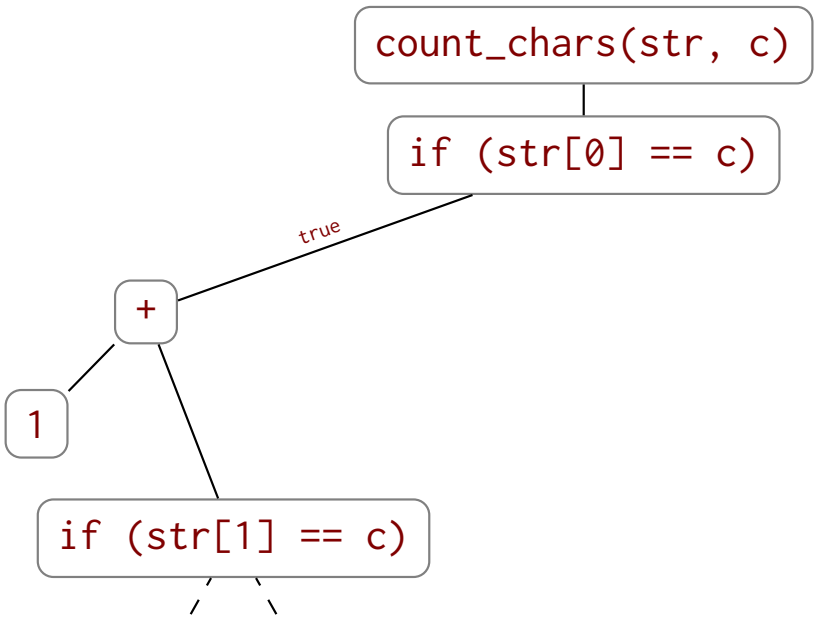
Implémentation sous la forme d'une expression (1)

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



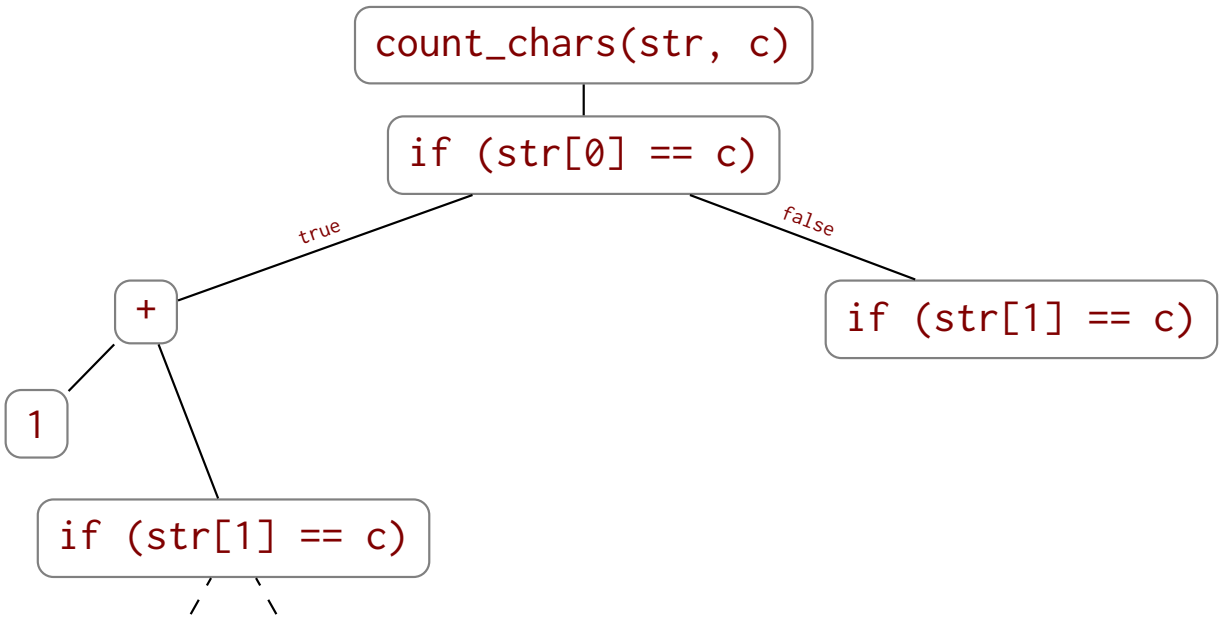
Implémentation sous la forme d'une expression (1)

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



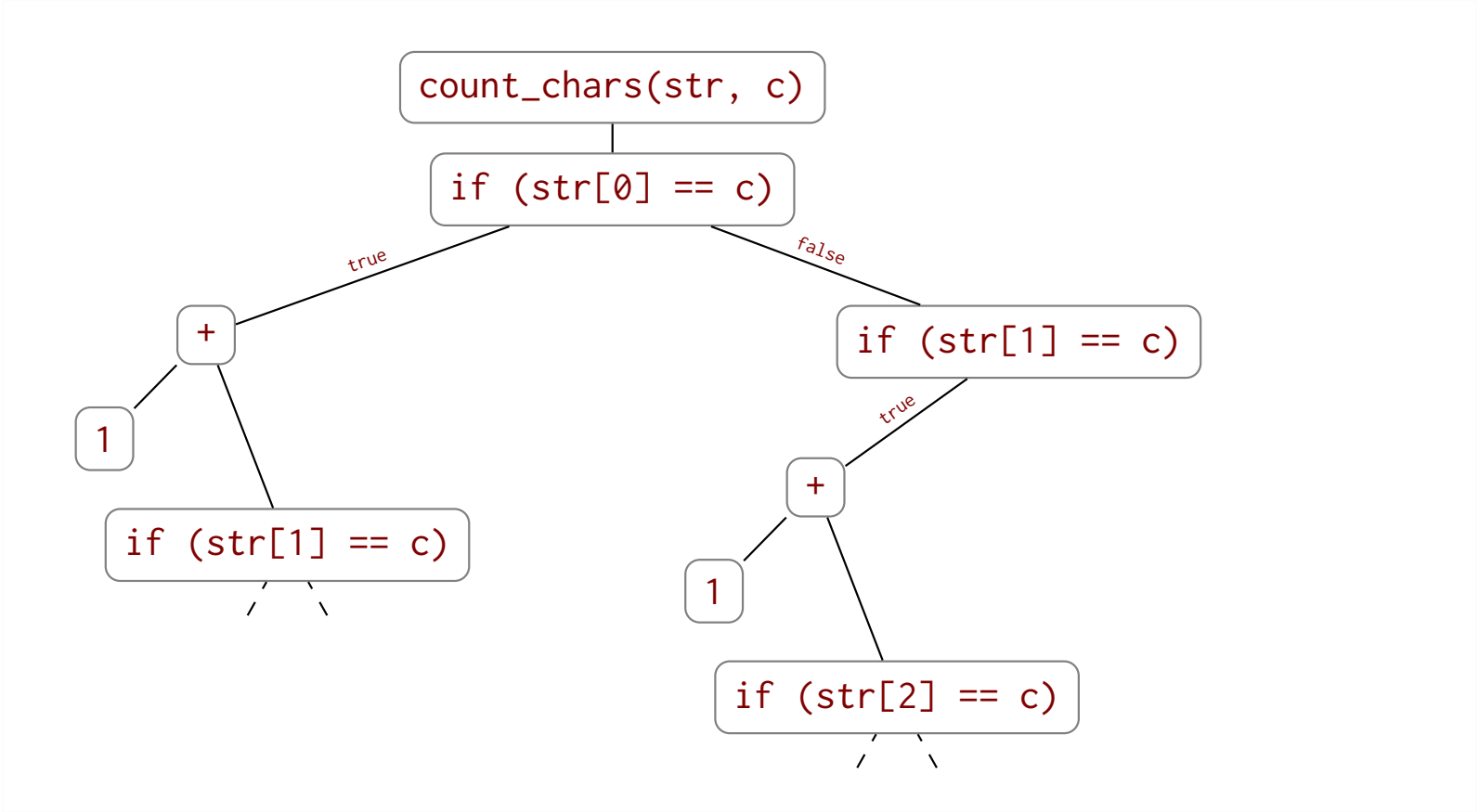
Implémentation sous la forme d'une expression (1)

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



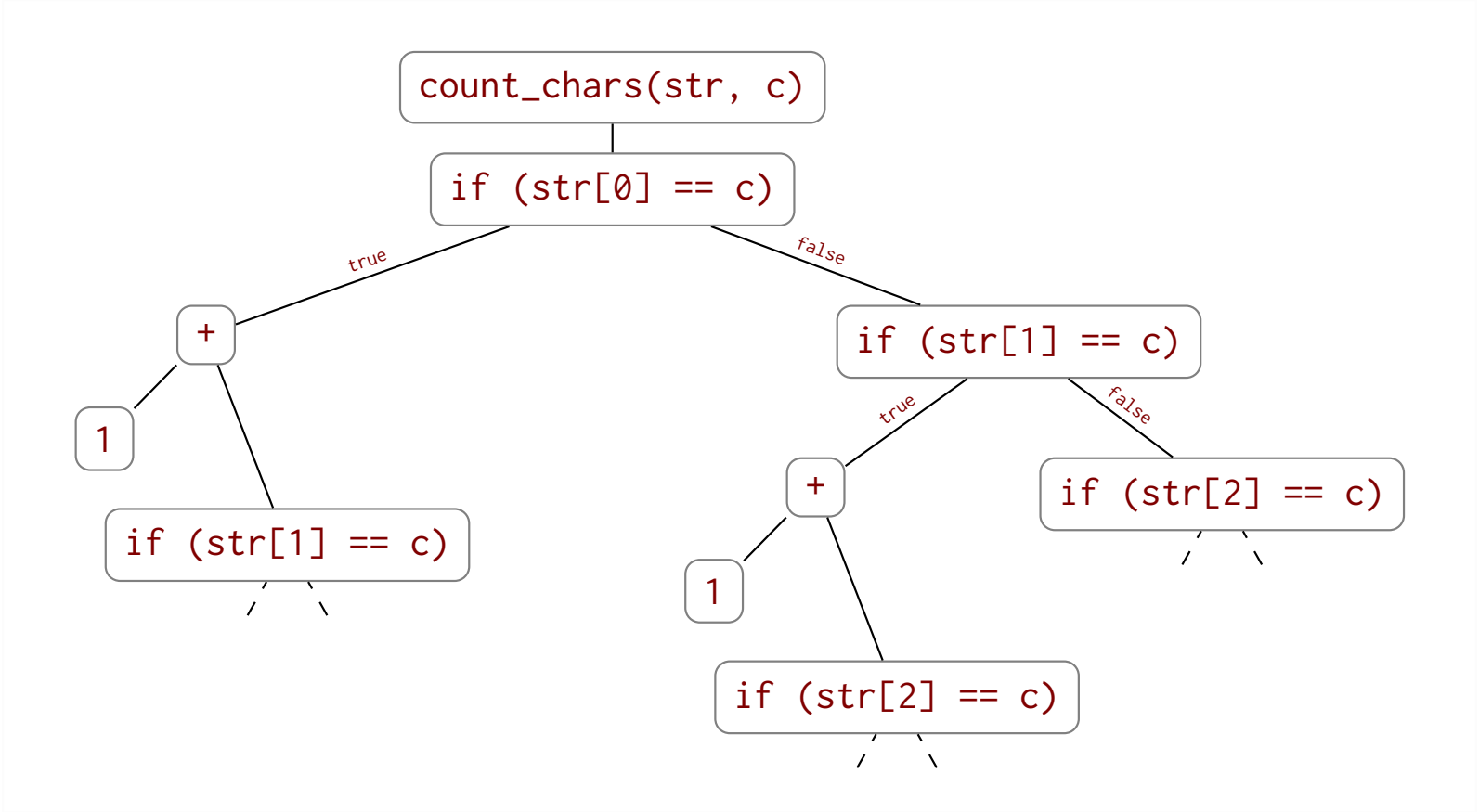
Implémentation sous la forme d'une expression (1)

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



Implémentation sous la forme d'une expression (1)

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



Quelques remarques :

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

Ces éléments nous invitent à résoudre le problème :

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

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

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

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

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

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

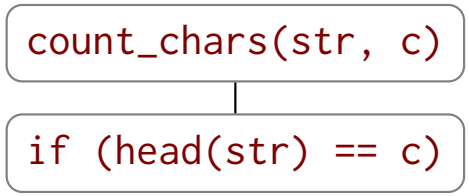
```
count_chars(str, c)
```

Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

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

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

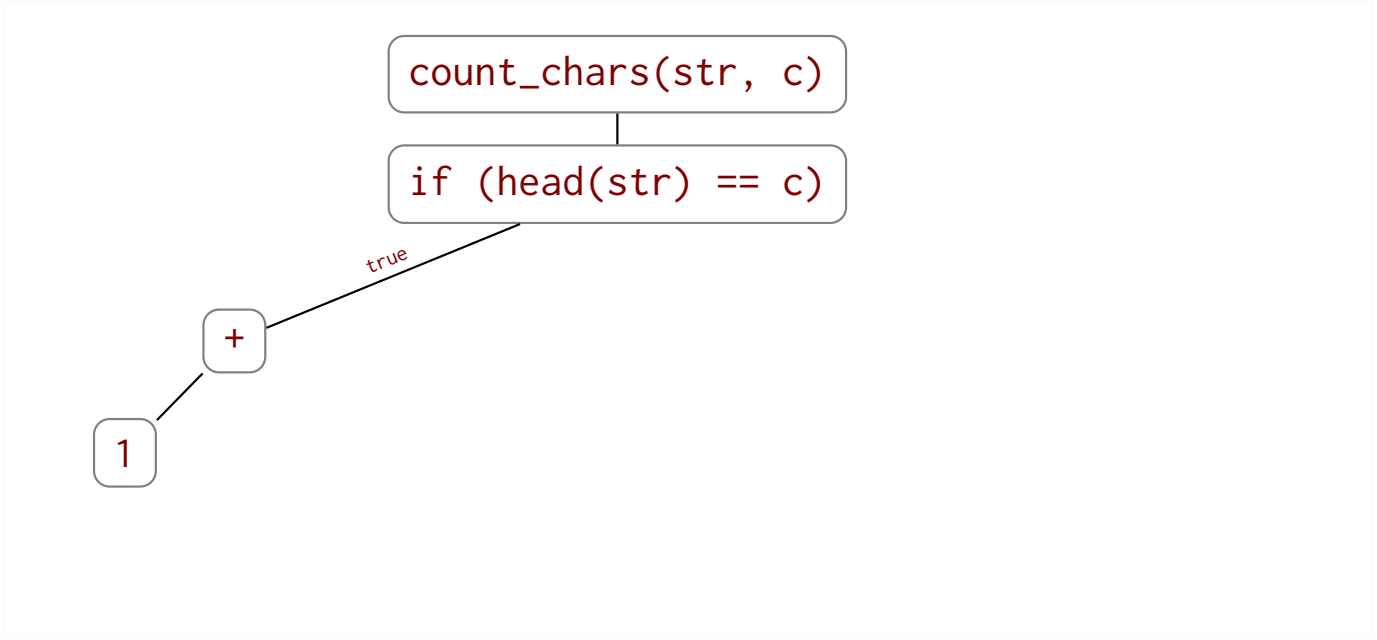


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

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

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

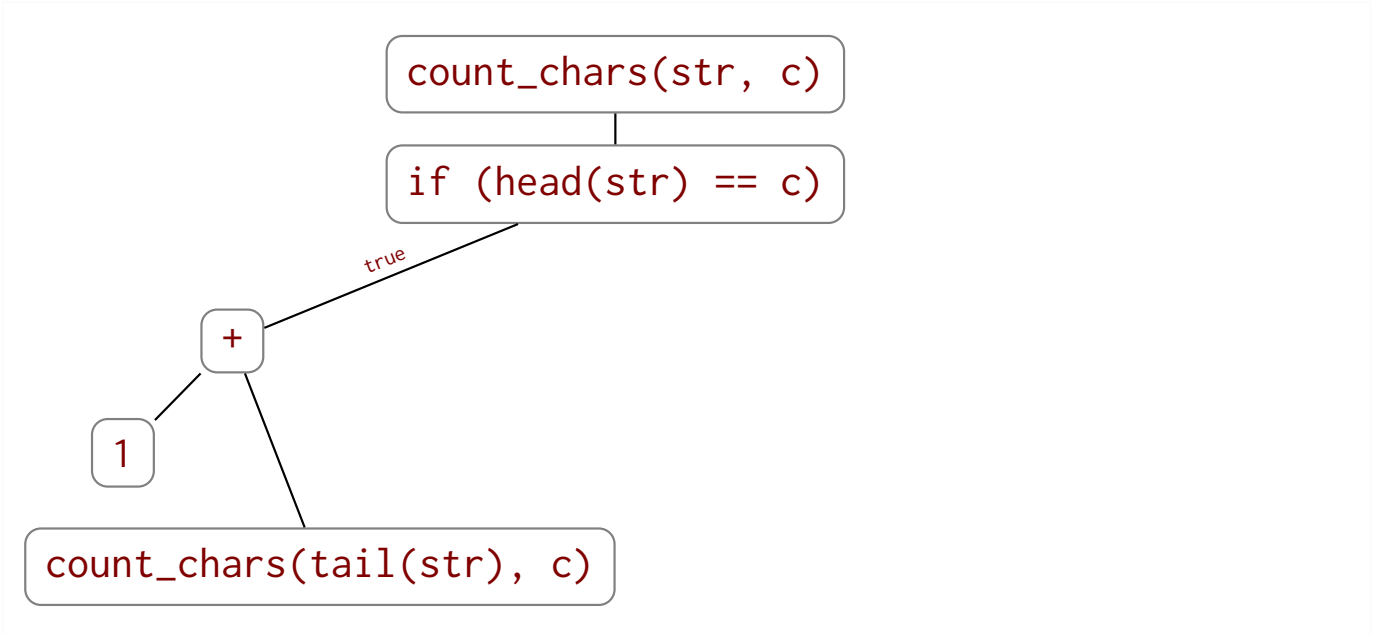


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

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

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

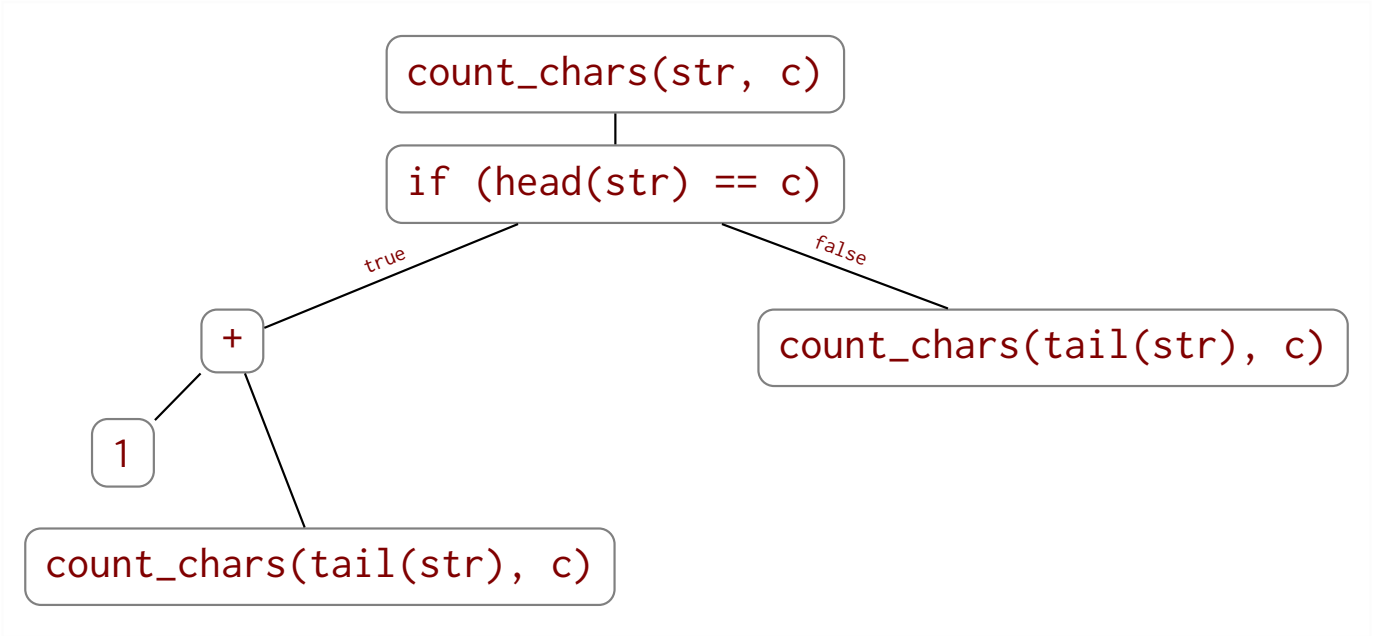


Implémentation sous la forme d'une expression (2)

Supposons disposer de deux opérations sur les chaînes de caractères :

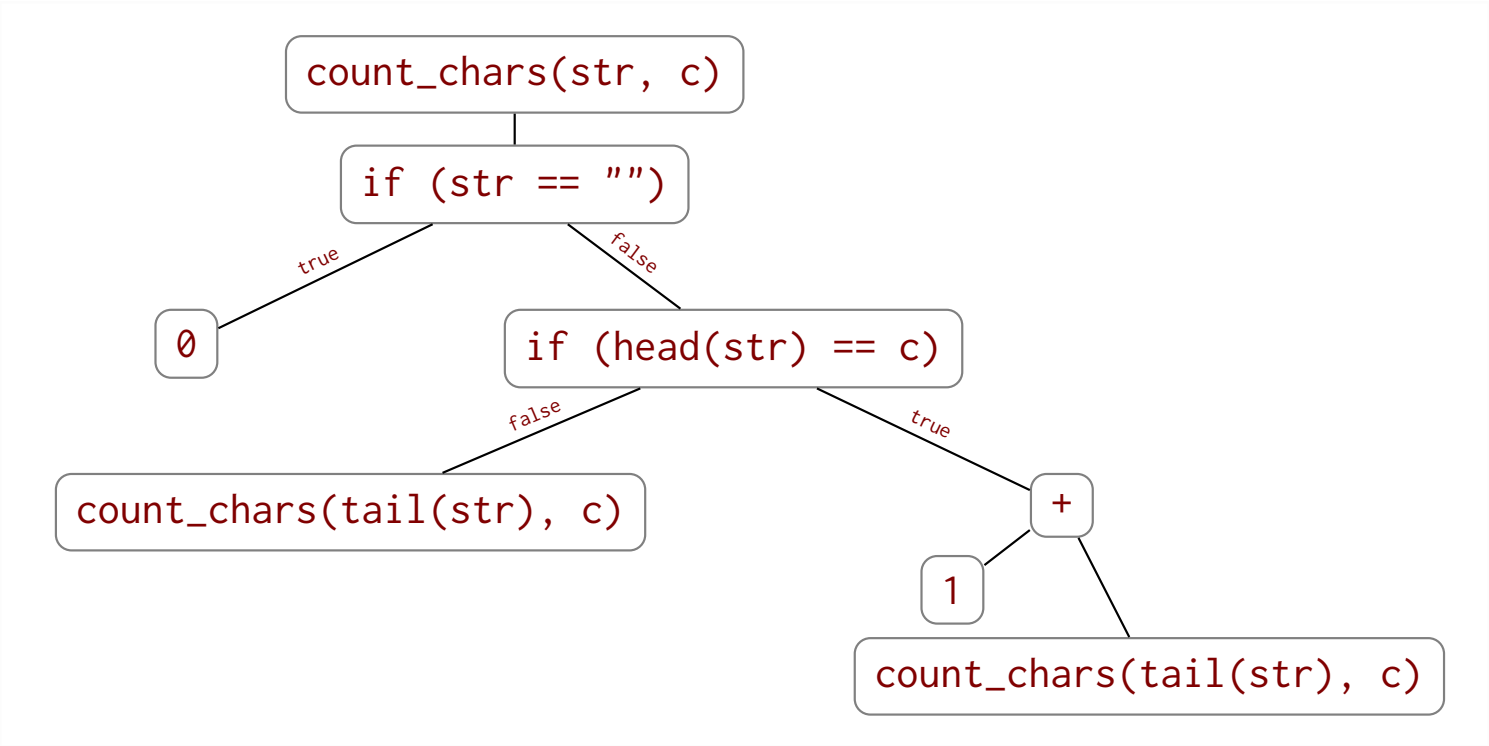
- une fonction `head(str)` qui renvoie le 1er caractère de `str` ;
- une fonction `tail(str)` qui renvoie une nouvelle chaîne de caractères correspondant à `str` privée de son 1er caractère.

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



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

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



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

Version finale Javascript

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

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

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

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

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

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

Comparaison des styles

Version impérative (I) :

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

Version fonctionnelle (F) :

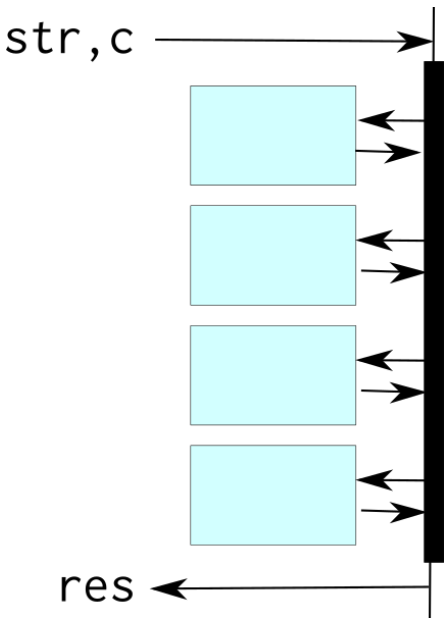
```
function countChars(str, c) {
  if (str.length === 0)
    return 0;
  else if (head(str) === c)
    return 1 + countChars(tail(str), c);
  else
    return countChars(tail(str), c);
}
```

Comparaison entre les versions :

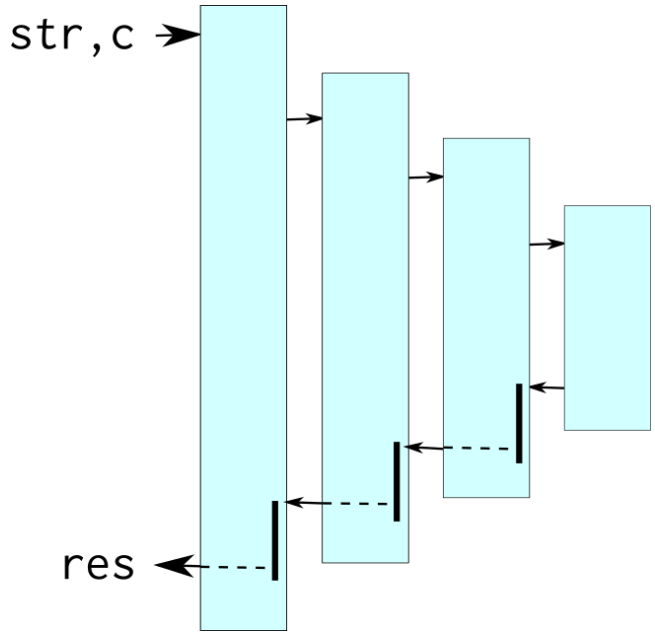
- **Abstraction** : la version (I) requiert une connaissance concrète de l'implémentation des chaînes de caractères, et de la façon d'itérer dessus ;
- **Indépendance** : les calculs de la version (F) n'interfèrent pas entre eux ; il n'y a pas d'état intermédiaire, explicite ou implicite.

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

Version impérative



Version fonctionnelle



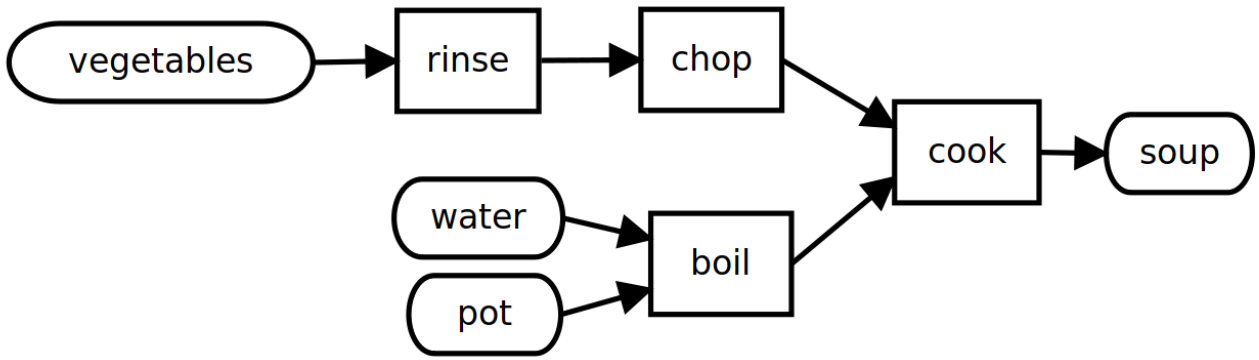
Aussi, le style fonctionnel promeut l'**indépendance** des calculs entre eux.

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

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

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

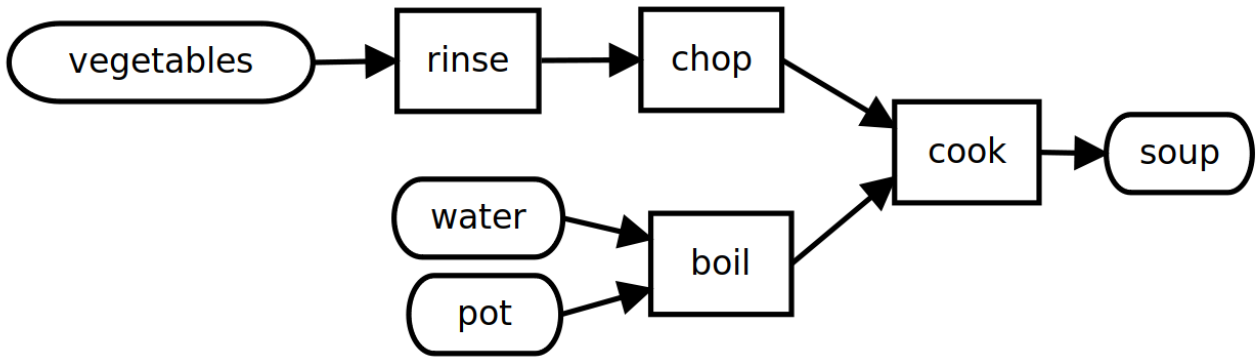
La description de l'algorithme suivante est elle impérative ou fonctionnelle ?



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

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

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



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

Réponse : **ni l'un ni l'autre**.

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

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

Version fonctionnelle

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

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

Version impérative

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

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

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

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

Une brève histoire de la programmation

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

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

Une brève histoire de la programmation

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

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

Une brève histoire de la programmation

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

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

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

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

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

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

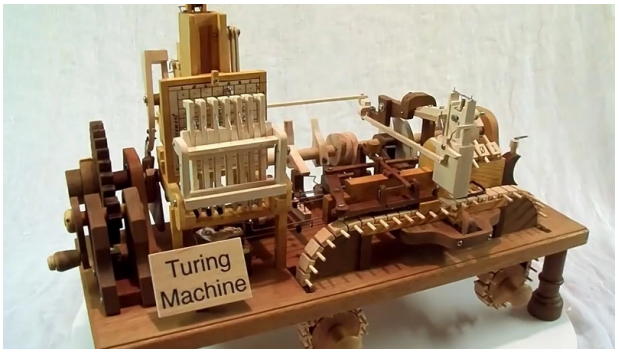
Le lambda-calcul

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

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

- 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

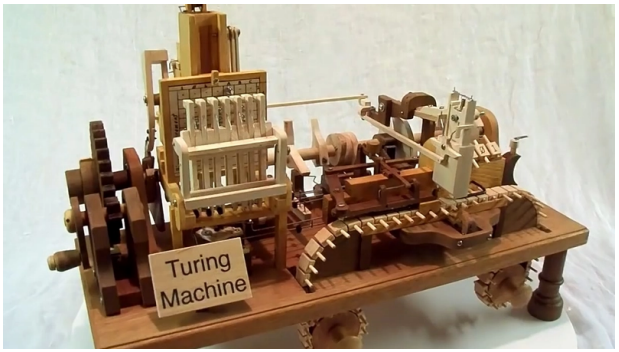
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

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

- 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

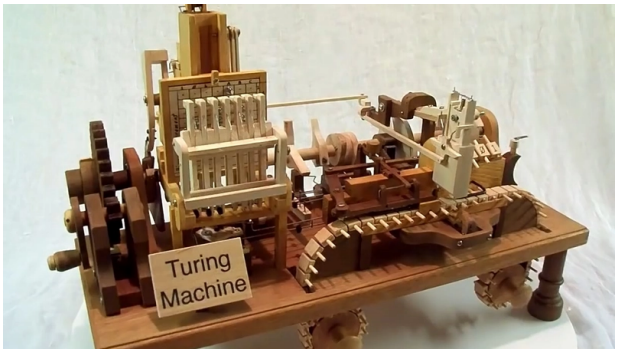
$$\texttt{true} ::= \lambda x.\ \lambda y.\ x$$
$$\texttt{false} ::= \lambda x.\ \lambda y.\ y$$
$$\texttt{if} ::= \lambda i.\ \lambda t.\ \lambda e.\ i\ t\ e$$
$$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ \texttt{false}\ \texttt{true}$$

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

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

"Style" fonctionnel

Les machines de Turing



Source : https://hackaday.com

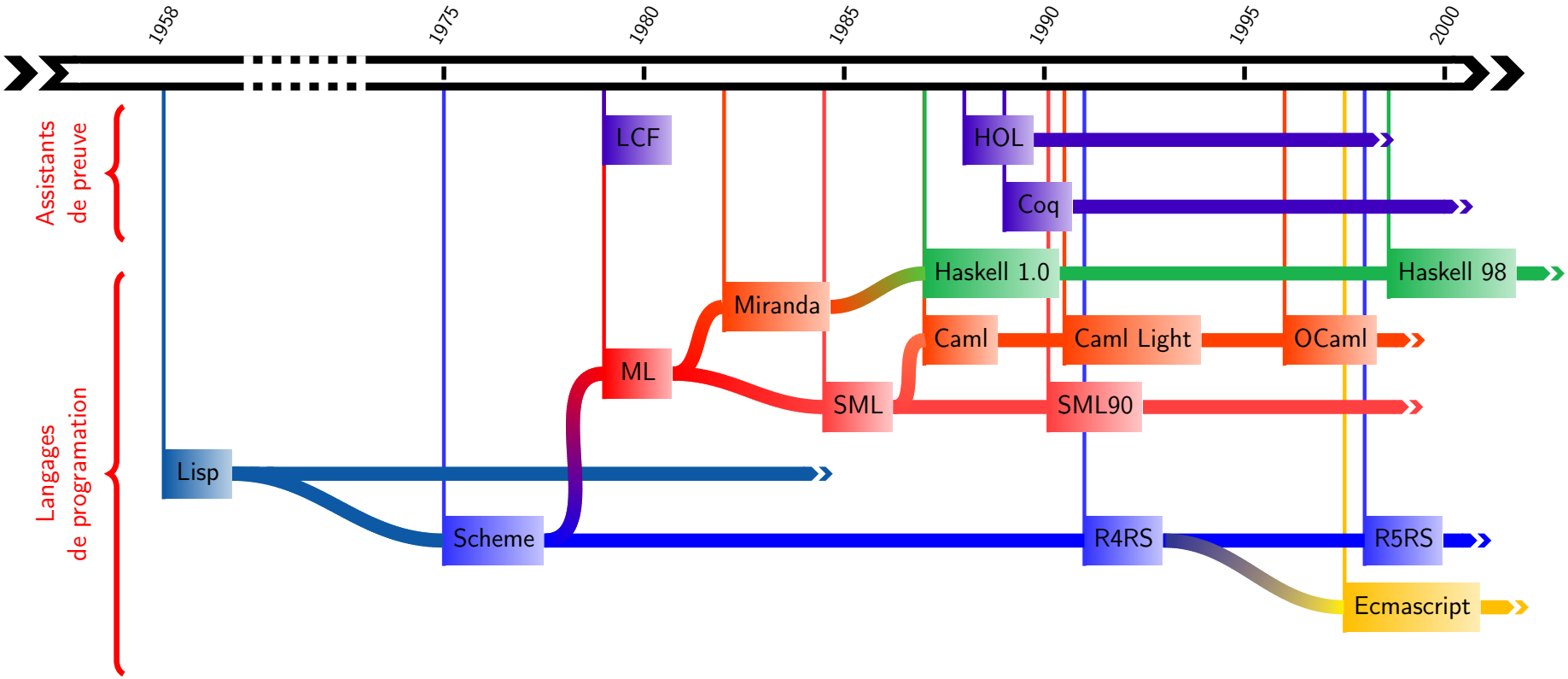
- Une machine formelle "concrète"
- Un état global modifié par des instructions

"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



Quelques lectures . . .

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

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

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

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

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

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