Programmation fonctionnelle et Javascript

D. Renault

GT Innovation pédagogique

31 mai 2023

La programmation fonctionnelle

Ça peut ressembler à cela :

. . . en Scheme

```
;; Merge two lists, assuming they are already sorted
(define (merge a b)
  (cond [(null? a) b]
        [(null? b) a]
        [(< (car a) (car b)) (cons (car a ) (merge (cdr a) b))]
        [else (cons (car b ) (merge a (cdr b)))]))

;; Merge sort implementation
(define (sort-merge l)
  (if (<= (length l) 1)
      l
      (let ([half (quotient (length l) 2)])
        (merge (sort-merge (take l half))(sort-merge (drop l half))))))
```

La programmation fonctionnelle

Ça peut ressembler à cela :

. . . en Haskell

```haskell
-- Merge two lists, assuming they are already sorted
merge [] ys          = ys
merge xs []          = xs
merge (x:xs) (y:ys) | x < y     = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys

-- Merge sort implementation
sort_merge []  = []
sort_merge [x] = [x]
sort_merge  xs = merge (sort_merge left) (sort_merge right)
        where (left,right) = (take lhx xs, drop lhx xs)
              lhx = length xs `div` 2
```

Petit historique (1)

A l'ENSEIRB, on a enseigné la programmation fonctionnelle en Lisp, puis en Scheme, pendant plus de temps que moi.
Ce sont de **bons** langages pour enseigner la programmation fonctionnelle (et en plus ils ont une histoire).
Problème : ils sont **singuliers** (par leur syntaxe, leurs spécificités, et par le fait qu'ils ne sont enseignés qu'à un unique moment de la formation).
Cela rend les idées dispensées dans le cours difficiles à appliquer ailleurs.

Petit historique (2)

Et en 2015, je suis invité à parler de programmation fonctionnelle en Javascript aux JDev.
C'est là que naît l'idée de choisir ce langage pour enseigner la programmation fonctionnelle.
La syntaxe est proche du langage C, enseigné de manière intensive à l'ENSEIRB-Matmeca en 1ère année.
Et le langage est suffisamment flexible pour y rendre la programmation fonctionnelle raisonnable.

C'est quoi la programmation fonctionnelle ?

Un style de programmation qui encourage la composition des fonctions (en opposition par exemple à la séquence d'instructions).
Deux exemples que je présente en introduction du cours.

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 $ax^2 + bx + c = 0$.

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

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 ?

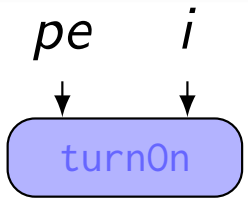
Un problème plus complexe (2/2)

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

Objectif

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

Un problème plus complexe (2/2)



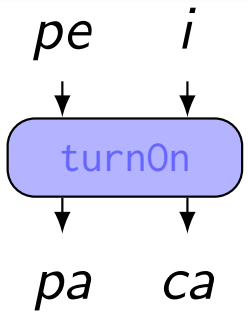
$$ca + pa = pe + i$$
$$ce = ca + s$$
$$c = b + pa + ce$$
$$po = pf + c$$

Objectif

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

Un problème plus complexe (2/2)

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

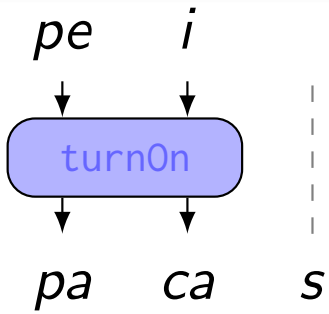


Objectif

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

Un problème plus complexe (2/2)

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

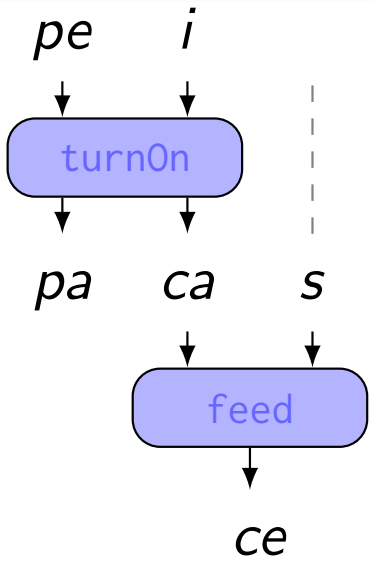


Objectif

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

Un problème plus complexe (2/2)

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

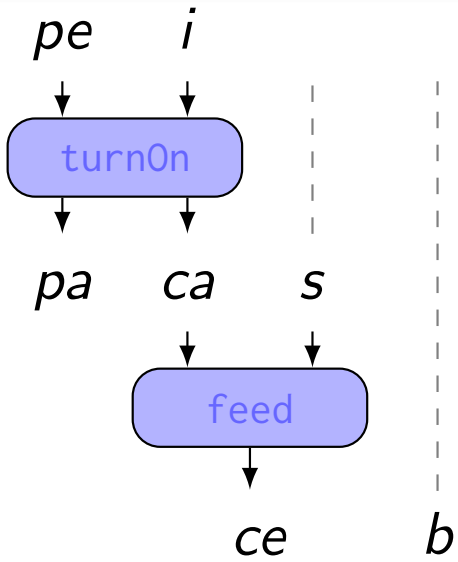


Objectif

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

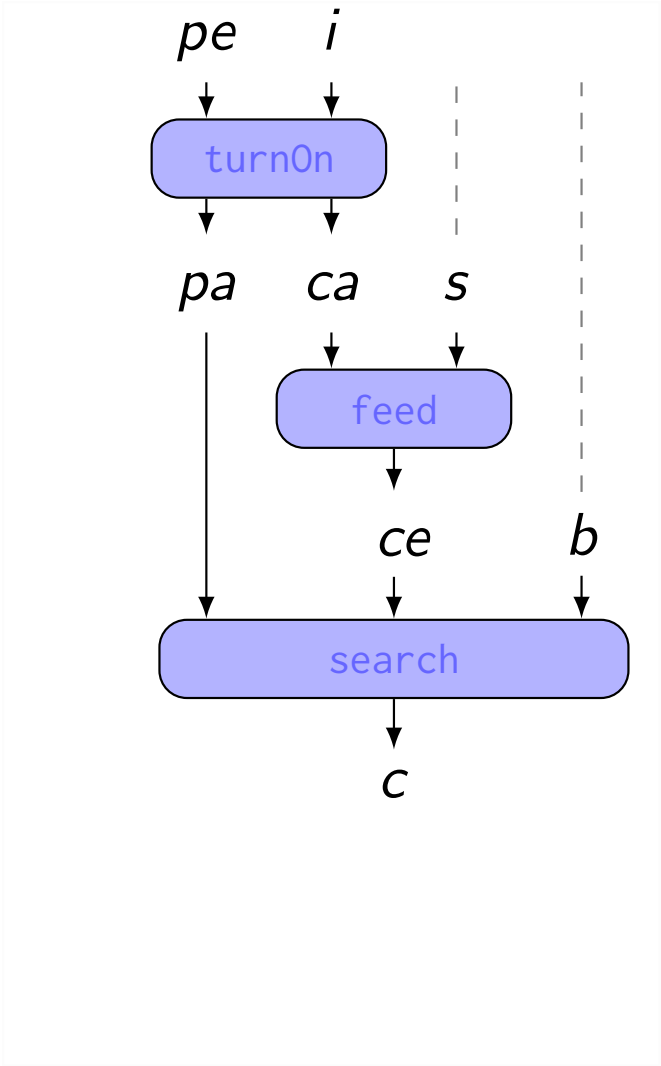
À partir de pe, pf, i, b, s, produire po.

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.

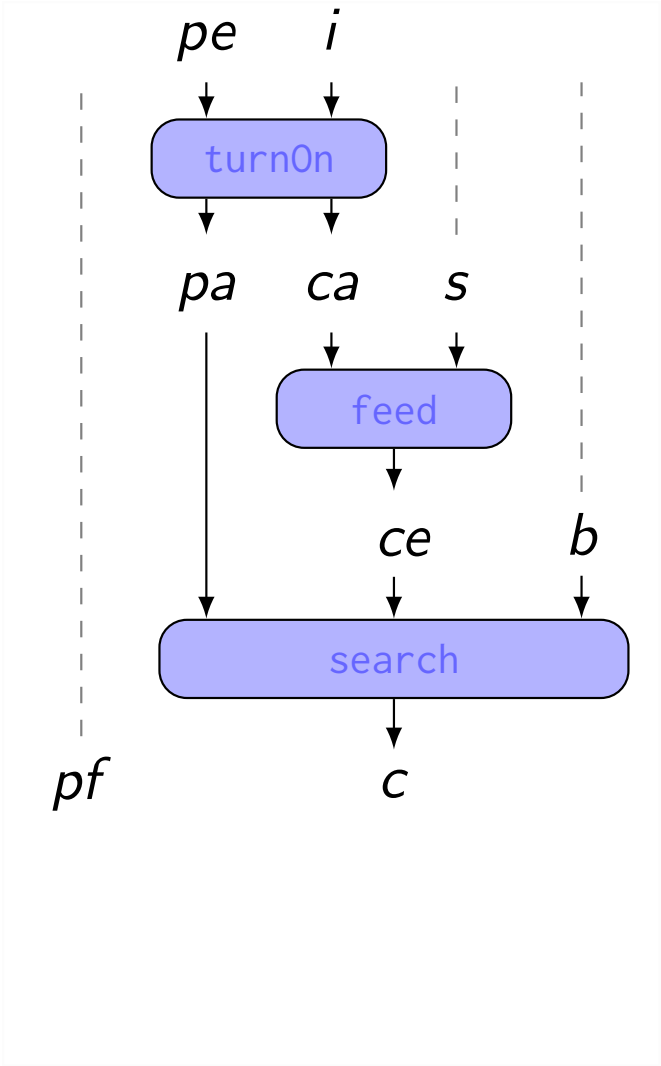


Un problème plus complexe (2/2)

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

Objectif

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

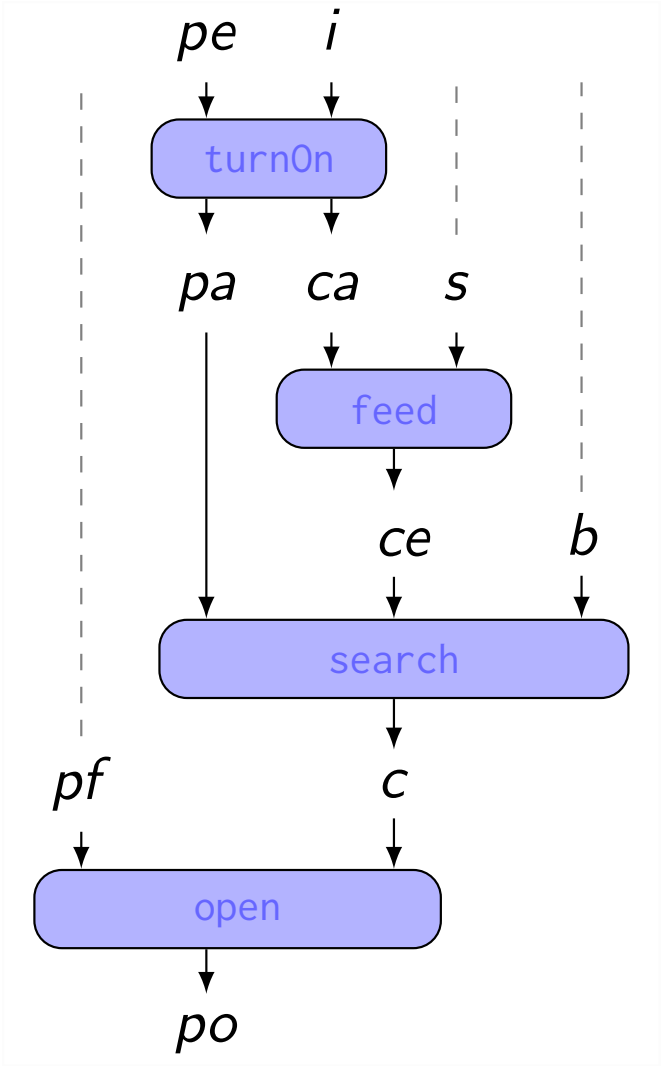


Un problème plus complexe (2/2)

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

Objectif

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



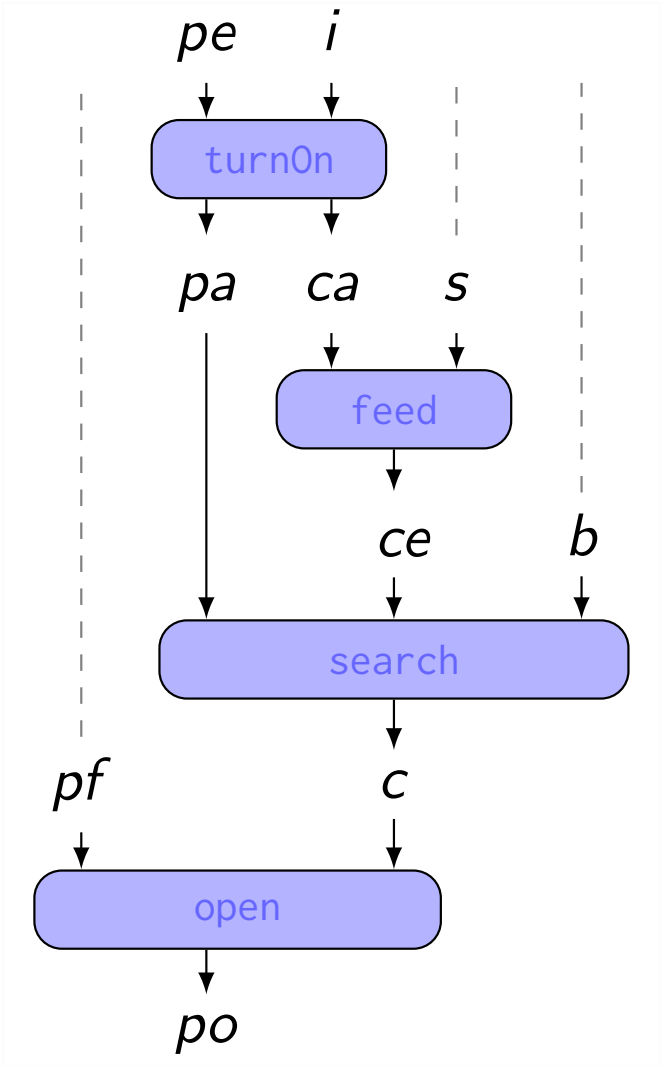
Un problème plus complexe (2/2)

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

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.

Revenons à la discussion générale

Idée

Voir la programmation fonctionnelle comme un **style** de programmation appliquable dans des situations variées, avec des **qualités** et des défauts.

Ceci à travers les deux grandes facettes de la programmation fonctionnelle :

- la **pureté** : immutabilité, dépendance, modularité, abstraction ...
- la **première classe** : généricité, réutilisation, extensibilité ...

Bref : montrer la programmation fonctionnelle en regard des autres styles de programmation.

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.

Le Javascript pour la programmation fonctionnelle

Qualités

- Syntaxe proche du C
- Fonctions et objets de première classe
- Présence de `const` et de méthodes pures

Défauts

- Tolérance considérable aux erreurs (i.e peu de vérification)
- Inspiration objet forte et singulière
- Pas d'optimisation des appels récursifs terminaux

Autres

- Orientation dynamique très forte

Exemple : méthodes pures et impures

Les manipulations de tableaux, des structures de données mutables, peuvent se faire au choix avec ou sans effet de bord.

- 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]
```

Exemple : la vérification du code (1/2)

L'Ecmascript est un langage particulièrement libéral :

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

D'où proviennent les problèmes dans les codes suivants ?

```
function cons(_car, _cdr)      { return { car: _car, cdr: _cdr }; }
function node(_val, _children) { return { val: _val, children: _children }; }

const aTree = node(7, cons(node(3, nil), cons(node(5, nil))));
const numC  = listLength(children(aTree)); // TypeError
```

Exemple : la vérification du code (1/2)

L'Ecmascript est un langage particulièrement libéral :

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

D'où proviennent les problèmes dans les codes suivants ?

```
function cons(_car, _cdr) { return { car: _car, cdr: _cdr }; }
function node(_val, _children) { return { val: _val, children: _children }; }

const aTree = node(7, cons(node(3, nil), cons(node(5, nil))));
const numC  = listLength(children(aTree)); // TypeError
```

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

Exemple : la vérification du code (1/2)

L'Ecmascript est un langage particulièrement libéral :

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

D'où proviennent les problèmes dans les codes suivants ?

```
function bizarreReturn() {
    return
           { car: 1 };
}
const strangeHead = bizarreReturn().car; // TypeError
```

Exemple : la vérification du code (1/2)

L'Ecmascript est un langage particulièrement libéral :

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

D'où proviennent les problèmes dans les codes suivants ?

```
function bizarreReturn() {
    return ;
           { car: 1 };
}
const strangeHead = bizarreReturn().car; // TypeError
```

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

Exemple : la vérification du code (2/2)

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

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

Exemple : ESLint

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

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

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

Exemple : la vérification du code (2/2)

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

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

Exemple : le compilateur Typescript

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

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

Typescript

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

Typescript

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

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

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

Typescript

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

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

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

Typescript

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

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

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

Les types et l'enseignement

Le mélange des deux langages offre la possibilité de tester différentes manières de faire de la **vérification**, et surtout de manière indépendante.

- les assertions
- les tests
- les types

Chaque style est optionnel, et donc abordable de manière **indépendante**.

L'écosystème Node

L'environnement Node.js (`https://nodejs.org`) utilisé 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 (en nombre gigantesque)

Avantages et inconvénients :

- Accès à de nombreux outils (Typescript, linter, tests . . .)
- Installation et portabilité aisée
- Problème : chaque répertoire de code contient rapidement 100Mo de bibliothèques

Node et programmation fonctionnelle

Exemples de bibliothèques encourageant la programmation fonctionnelle :

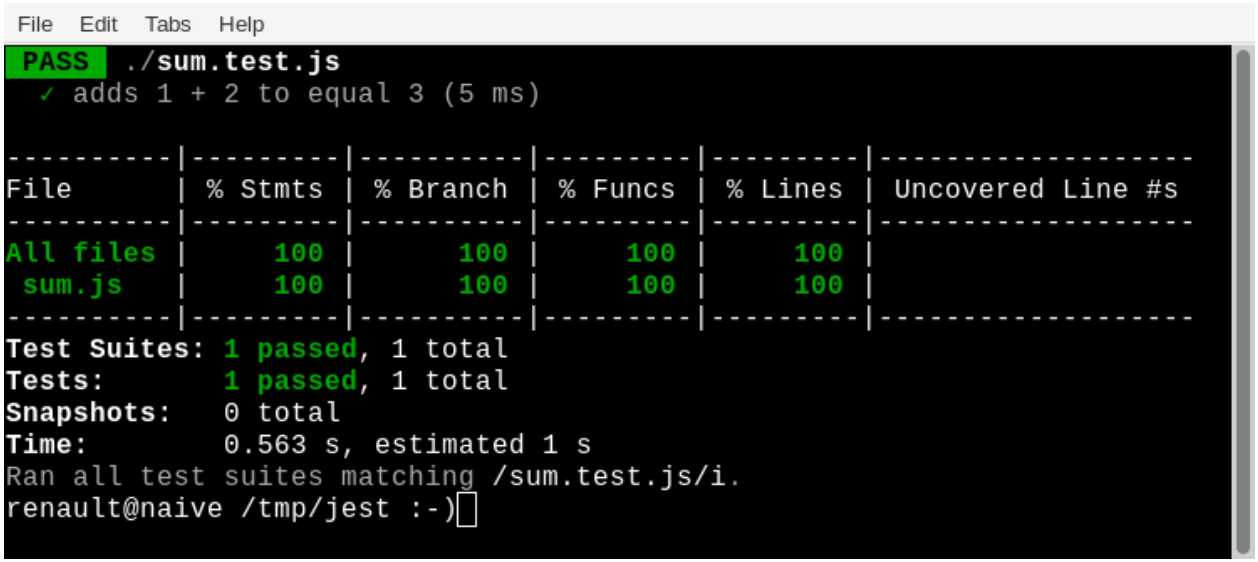
- underscore (https://underscorejs.org)
- ramda (https://ramdajs.com)
- lodash (https://lodash.com)
- immutable-js (https://immutable-js.com)
- list (https://github.com/funkia/list)

Node et les tests

Jest https://jestjs.io est une bibliothèque de test riche, facilement installable et avec des petits morceaux de programmation fonctionnelle :

```
const S = require('./sum');

test('adds 1 + 2 to equal 3', () => {    // A lambda
  expect(S.sum(1, 2)).toBe(3);           // A matcher
});
```



Prog. fonctionnelle et tests automatiques

Chaque élève dispose d'un dépôt git

Il programme dans son environnement à lui

Les tests sont exécutés à chaque commit

Prog. fonctionnelle et examen machine

L'examen final est réalisé sur machine

- environnement identique à celui des TDs
- accès limité au réseau
- présence des bibliothèques et des outils (tsc, eslint . . .)