Programmation fonctionnelle

Guy Tremblay
Professeur titulaire
Département d'informatique

http://www.labunix.uqam.ca/~tremblay_gu

Webinaire Latece
23 juin 2020

UQÀM

AUGUST 16, 2016

The rise of functional programming & the decline of Angular 2.0

Learn why we believe that the rise of functional programming (FP) will prevent Angular 2.0 from being as successful as its predecessor.

Functional Programming is on the rise

Roman Elizarov Follow

May 4, 2019 · 5 min read

From my perspective, yes. Many companies are investing in functional programming. Facebook created React as a functional frontend view framework. They've also built Immutable.js, a library of immutable data structures inspired by Clojure's. Big tech companies like Facebook, Twitter, Amazon, and Paypal use functional languages like Erlang, Scala, Haskell, and Clojure. Walmart, Staples, and Monsanto have Clojure divisions. Google, Target, Intel, and Microsoft use Haskell. **These giant companies are not letting functional languages go anywhere.** Meanwhile, functional programming conferences are popping up everywhere.

3. Programmers Will Become More Dependent on Functional Languages

The concept supports running sections of software in parallel, across different machines and CPU cores. That eliminates the need for complex synchronization. Therefore, Web requests and other functions requiring concurrent processing can be better managed. The trends also affect programmers of smartphone applications, interconnected devices, and servers that support the interactions between them.

Languages such as Scala, Haskell, and Clojure have not only grown in popularity but are expected to continue to as the demands for flexible, functional programming expand.

ACM NEWS

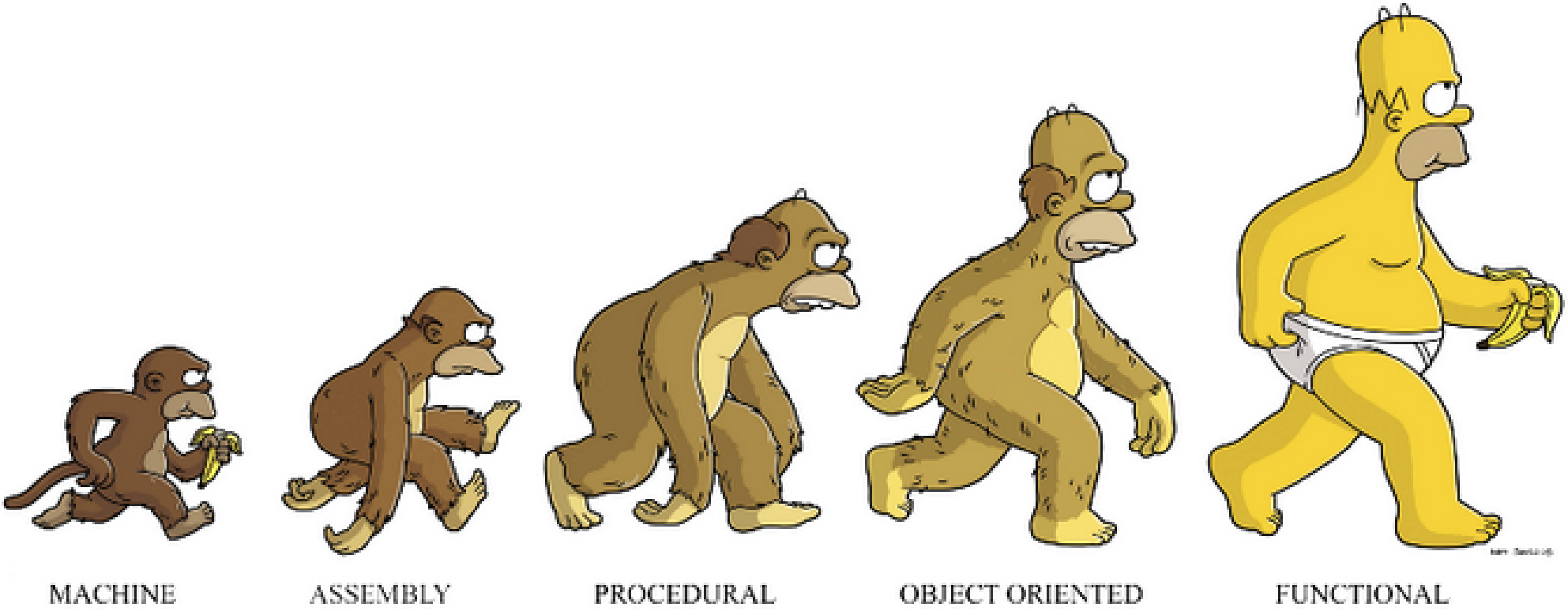
What's the Future of Programming? The Answer Lies in Functional Languages

By TechRepublic
October 30, 2017

Why Functional Programming Is on the Rise Again

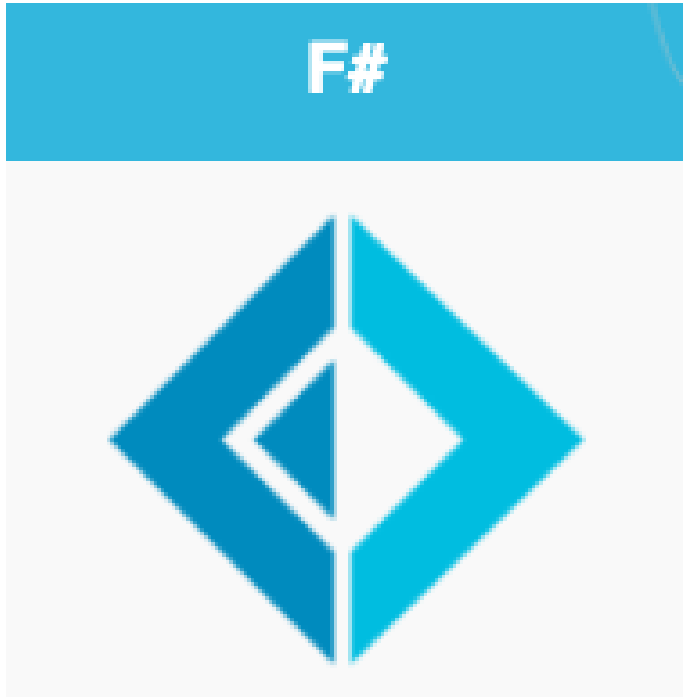
by **Tom Helvick** | Dec 12, 2018

L’évolution des paradigmes de programmation



2005

2005



2011

2012

TREMBLAY*GUY

Le dossier est archivé.

baccalauréat en mathématiques

3	MAT3070	1	A	3c	813	MAT3112	1	A	3c
1	MAM5160	1	S	3c	821	MAT2741	1	A	3c
1	PHY2621	1	A	3c	823	MAT2110	1	A	3c

MAT2741 : LISP et ses applications


STEVEN SPIELBERG PRESENTS
BACK TO THE FUTURE
PG
A ROBERT ZEMECKIS FILM


WHATEVER HAPPENS MARTY DON'T GO TO 2020!

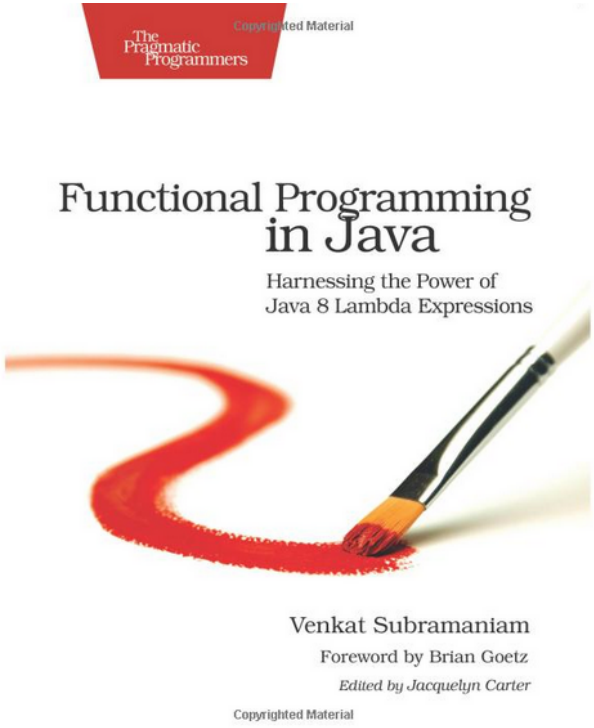
Programmation fonctionnelle : «Retour vers le futur» !?

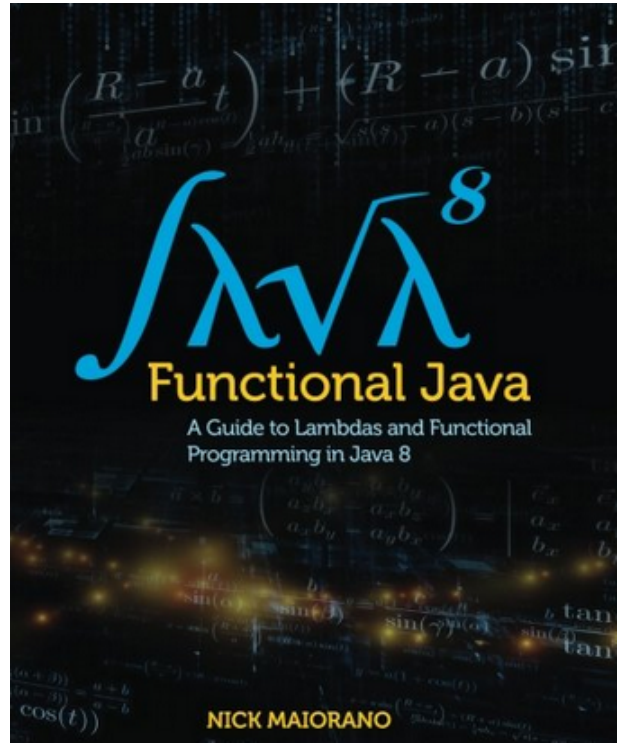
Guy Tremblay
Professeur titulaire
Département d'informatique

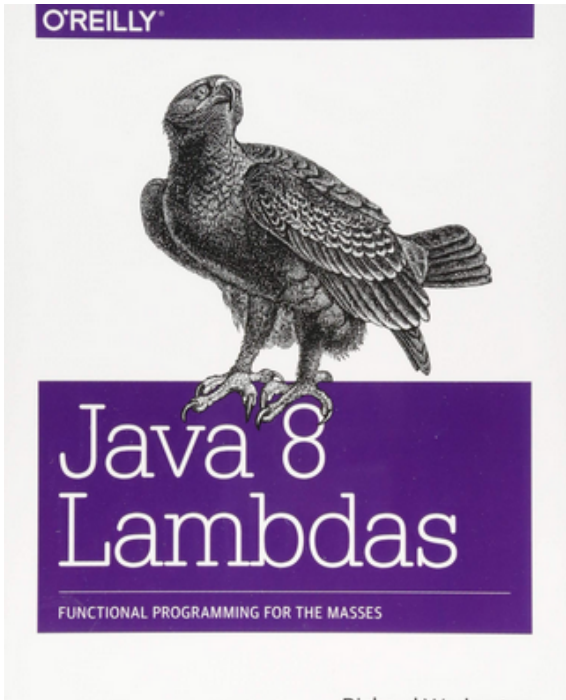
http://www.labunix.uqam.ca/~tremblay_gu

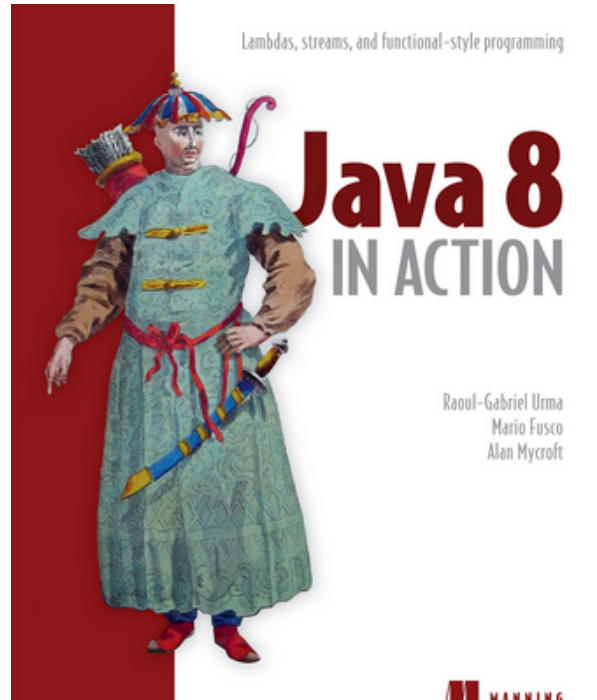
Webinaire Latece
23 juin 2020

UQÀM


The Pragmatic Programmers
Copyrighted Material
Functional Programming in Java
Harnessing the Power of Java 8 Lambda Expressions
Venkat Subramaniam
Foreword by Brian Goetz
Edited by Jacquelyn Carter
Copyrighted Material


Functional Java
A Guide to Lambdas and Functional Programming in Java 8
NICK MAIORANO


O'REILLY
Java 8 Lambdas
FUNCTIONAL PROGRAMMING FOR THE MASSES


Lambdas, streams, and functional-style programming
Java 8
IN ACTION
Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

Quelques jalons dans l'évolution de Java depuis 2004

Types génériques	Java 5 (2004)	
`Future`		
Expressions lambdas (avec/sans types explicites)	Java 8 (2014)	
`Streams`		
`CompletableFuture`		
Mot-clé `var` pour variables locales $\Rightarrow$Type implicite	Java 10 (2018)	
Mot-clé `var` pour arguments des λ-expr. $\Rightarrow$ Type implicite	Java 11 (2018)	
Expressions `switch` étendues	Java 12 (2019)	
Pattern-matching	Java 14 (2020)	

Quelques jalons dans l'évolution de Java depuis 2004

Types génériques	Java 5 (2004)	ML (1973)
`Future`		MultiLisp (1985)
Expressions lambdas (avec/sans types explicites)	Java 8 (2014)	Lisp (1958)
`Streams`		Lisp (1958)
`CompletableFuture`		Id (1987)
Mot-clé `var` pour variables locales $\Rightarrow$Type implicite	Java 10 (2018)	ML (1973)
Mot-clé `var` pour arguments des λ-expr. $\Rightarrow$ Type implicite	Java 11 (2018)	ML (1973)
Expressions `switch` étendues	Java 12 (2019)	Lisp (1958)
Pattern-matching	Java 14 (2020)	Hope (1980)

Une citation de C.A.R. Hoare

«[ALGOL-60] is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.»

C.A.R. Hoare

Pour paraphraser C.A.R. Hoare

«[Lisp, ML, Miranda are] language[s] so far ahead of [their] time, that [they were] not only an improvement on [their] predecessors, but also on nearly all [their] successors.»

G. Tremblay

Programmation fonctionnelle : Quel héritage les langages fonctionnels nous ont-ils légué ?

Guy Tremblay
Professeur titulaire
Département d'informatique

http://www.labunix.uqam.ca/~tremblay_gu

Webinaire Latece
23 juin 2020

UQÀM

Aperçu

Aperçu

Il existe plus de 7 000 langues (*spoken languages*)

Question* : Combien existe-t-il de langages de programmation ?

Il existe plus de 7 000 langues (*spoken languages*)

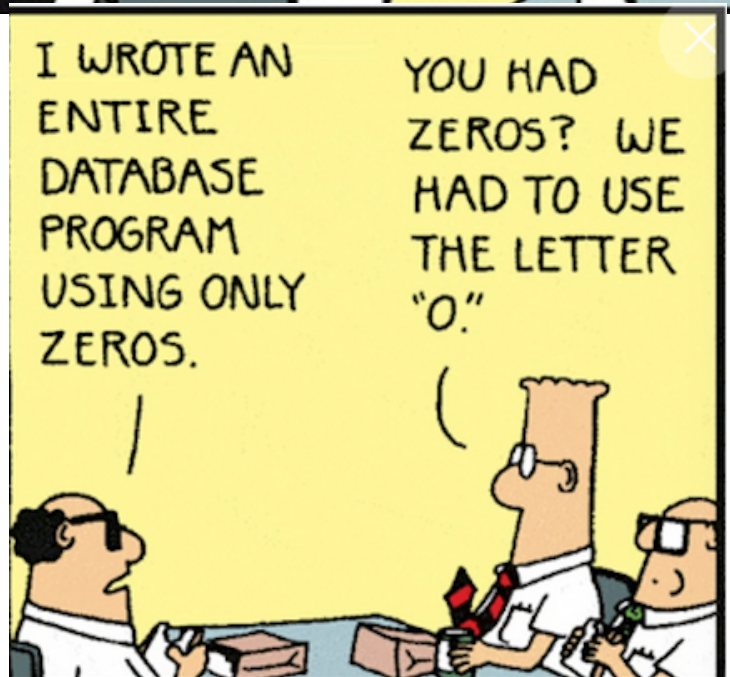
Question* : Combien existe-t-il de langages de programmation ?

Réponses :

- TIOBE : $\approx$ 250 (*popular programming languages*)
- Wikipedia : 700 (*«all notable prog. lang. in existence»*, pas *markup*)
- FOLDOC : 1 000 (années 90)
- The Language List : 2 500
- HOPL : 8 945
- CodeLani : 4 157 (*currently actively tracking*)

`https://codelani.com/posts/how-many-programming-languages-are-there-in-the-world.html`

Les langages de programmation sont nombreux et variés



Les langages de programmation sont nombreux et variés

Diverses versions d'un programme «Hello world !»

Brainfuck

```
++++++++++[>+++++++>++++++++++>+++>\
+<<<<-]>++.>+.+++++++..+++.>++.<<\
+++++++++++++++.>.+++.------.\
--------.>+.>.
```

Source: `http://esolangs.org/wiki/Hello_world_program_in_esoteric_languages`

Les langages de programmation sont nombreux et variés

Diverses versions d'un programme «Hello world !»

Verbose

```
PUT THE NUMBER LXXII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CI ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CVIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CXI ONTO THE TOP OF THE PROGRAM STACK
...
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER CVIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER C ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
PUT THE NUMBER XXXIII ONTO THE TOP OF THE PROGRAM STACK
GET THE TOP ELEMENT OF THE STACK AND CONVERT IT TO AN ASCII CHARACTER
AND OUTPUT IT FOR THE CURRENT PERSON USING THIS PROGRAM TO SEE
```

Source: `http://esolangs.org/wiki/Hello_world_program_in_esoteric_languages`

Les langages de programmation sont nombreux et variés

Diverses versions d'un programme «Hello world !»

Emoji code (`http://www.emojicode.org/`)

Source: `https://gist.github.com/EranAvidor/0b9a491bbe365bbf2cbe`

Les langages de programmation sont nombreux et variés

Diverses versions d'un programme «Hello world !»

Anguish

Here's an Anguish program that prints `Hello World`:

```
```

Dans ce qui suit. . .

- Il y a aura des exemples dans quelques langages, principalement fonctionnels, mais pas exclusivement

Question* : Quels sont les langages que vous avez utilisés ou que vous connaissez qui sont « fonctionnels» ?

Question* : Quels sont les langages que vous avez utilisés ou que vous connaissez qui sont « fonctionnels» ?

Que j'ai utilisés	Que je n'ai pas utilisés
Lisp	Scheme
Miranda	KRC/SASL
	FP
	Hope
Id	SISAL
Haskell	ML
pH	F#
Elixir	Erlang
Elm	Clojure

Question* : Quels sont les langages que vous avez utilisés ou que vous connaissez qui sont «purement fonctionnels» ?

Que j'ai utilisés	Que je n'ai pas utilisés
Lisp	Scheme
Miranda	KRC/SASL
	FP
	Hope
Id	SISAL
Haskell	ML
pH	F#
Elixir	Erlang
Elm	Clojure

Question* : Quels sont les langages que vous avez utilisés ou que vous connaissez qui sont «purement fonctionnels» ?

Que j'ai utilisés	Que je n'ai pas utilisés
Lisp	Scheme
Miranda	KRC/SASL
	FP
	Hope
Id	SISAL
Haskell	ML
pH	F#
Elixir	Erlang
Elm	Clojure

Dans ce qui suit. . .

- Il y a aura des exemples dans *quelques* langages, *principalement fonctionnels*, mais pas exclusivement

- Nous allons examiner. . .
  - les caractéristiques fondamentales des langages fonctionnels

  ET
  - certaines fonctionnalités intéressantes de langages fonctionnels qu'on retrouve dans divers autres langages

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

- Mécanismes d'abstraction et de modularisation = Fonctions, fonctions, . . . , fonctions

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

- Mécanismes d'abstraction et de modularisation = Fonctions, fonctions, . . . , fonctions

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• Yes, functions
• Strategy pattern	• Oh my, functions again!
• Decorator pattern	• Functions
• Visitor pattern	• Functions

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

- Mécanismes d'abstraction et de modularisation = Fonctions, fonctions, . . . , fonctions

  Fonctions = entités de 1ère classe

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

- Mécanismes d'abstraction et de modularisation = Fonctions, fonctions, . . . , fonctions

  Fonctions = entités de 1ère classe

- Manipulation de valeurs immuables

Quelles sont deux caractéristiques fondamentales d'un langage fonctionnel ?

- Mécanismes d'abstraction et de modularisation = Fonctions, fonctions, . . . , fonctions

  Fonctions = entités de 1ère classe

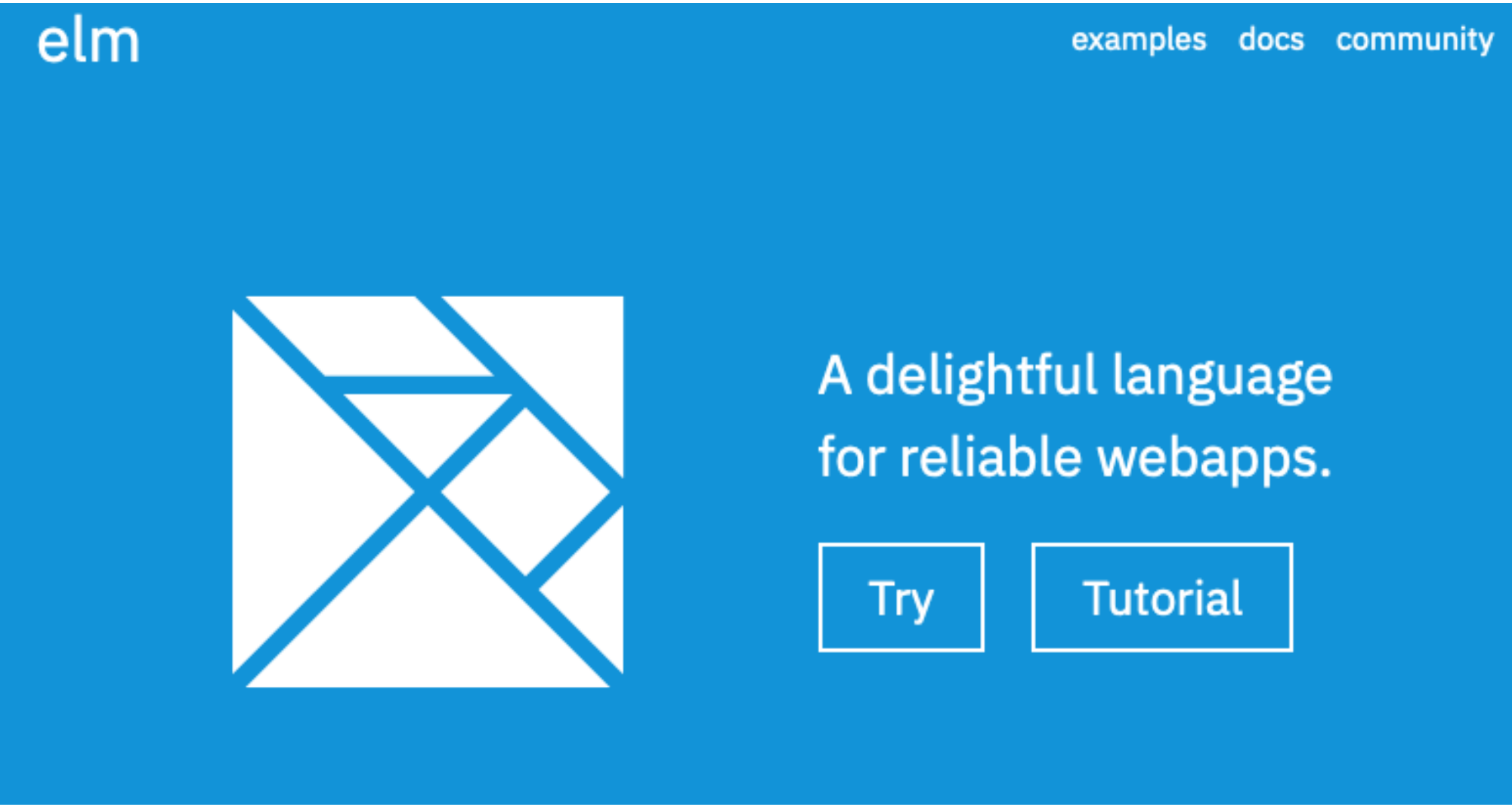
- Manipulation de valeurs immuables

  Pas de «variables» *mutables*
  OU
  Manipulation restreinte et limitée de variables mutables

1.1 Fonctions, fonctions, . . . , fonctions

Plusieurs exemples seront en `elm`

2https://elm-lang.org/



Elm (`elm repl`)

```
> plus x y = x + y
<function> : number -> number -> number

> plus
<function> : number -> number -> number

> plus 2 5
7 : number

> plus 2  -- Application partielle
<function> : number -> number
```

Elm

```
> liste = [1,1,2,3,5]
[1,1,2,3,5] : List number

> map
<function> : (a -> b) -> List a -> List b

> map (plus 2) liste
[3,3,4,5,7] : List number
```

Elm

```
> (+) -- Version prefixe d'un operateur infixe
<function> : number -> number -> number

> (+) 2
<function> : number -> number

> map ((+) 2) liste
[3,3,4,5,7] : List number
```

Elm

```
> applyAll fs x =  map (\f -> f x) fs
<function> : List (a -> b) -> a -> List b

> applyAll [(+) 2, always 2, abs]
<function> : number -> List number

> applyAll [(+) 2, always 2, abs] -8
[-6,2,8] : List number
```

Elm

```
> filter
<function> : (a -> Bool) -> List a -> List a

> isEven x = x % 2 == 0
<function> : Int -> Bool

> filter isEven [1, 1, 2, 3, 5]
[2] : List Int
```

Elm

```
-- Style applicatif (par ex., Python)
> filter isEven (map (plus 1) [1, 1, 2, 3, 5])
[2,2,4,6] : List Int
```

Elm

```
-- Style applicatif (par ex., Python)
> filter isEven (map (plus 1) [1, 1, 2, 3, 5])
[2,2,4,6] : List Int


-- Style pipeline (≈ Unix): F#, Elixir, Elm
> [1, 1, 2, 3, 5]
    |> map (plus 1)
    |> filter isEven
[2,2,4,6] : List Int
```

C : On peut aussi avoir des fonctions d'ordre supérieur en C !

```c
int plusDeux( int x ) { return x + 2; }

int* map( int (*f)(int), int a[], int nb )
    int* r = (int*) calloc( nb, sizeof(int) );

    for ( int i = 0; i < nb; i++ ) {
        r[i] = f(a[i]);  // Ou bien: (*f)(a[i]);
    }
    return r;
}


// APPEL
int a[] = // ... NB_ELEMENTS ...

int* r = map( plusDeux, a, NB_ELEMENTS );
```

JavaScript (node) : Ou en JavaScript

```
> function plusUn( x ) { return x + 1;}
undefined

> plusUn
[Function: plusUn]

> [1, 1, 2, 3, 5].map
[Function: map]

> [1, 1, 2, 3, 5].map( plusUn )
[ 2, 2, 3, 4, 6 ]

> [1, 1, 2, 3, 5].
... map( plusUn ).
... filter( (n) => n % 2 == 0 )
[ 2, 2, 4, 6 ]
```

Question* : Est-ce que `JavaScript` est un langage **fonctionnel** ?

Question* : Est-ce que `JavaScript` est un langage **fonctionnel** ?

Réponse : Pas vraiment !

1.2 Immutabilité

JavaScript (node)

```
> v = 10;
10

> function foo( x ) { v += 1; return x + v; }
undefined

> foo(11) // Effet de bord sur v!
22

> foo(11)
23

> foo(11) == foo(11)
false
```

JavaScript (node)

```
> v = 10;
10

> function foo( x ) { v += 1; return x + v; }
undefined

> foo(11) // Effet de bord sur v!
22

> foo(11)
23

> foo(11) == foo(11)
false
```

On veut des fonctions, oui, mais des «vraies» fonctions !

Des fonctions, oui, mais des «vraies» fonctions

Des fonctions. . . au sens «mathématique» du terme !

Fonction pure

Le résultat de l'appel à une fonction dépend uniquement des arguments fournis

$\Rightarrow$ Les mêmes arguments produisent le même résultat

Des fonctions, oui, mais des «vraies» fonctions

Des fonctions. . . au sens «mathématique» du terme !

Fonction pure

Le résultat de l'appel à une fonction dépend uniquement des arguments fournis

- $\Rightarrow$ Les mêmes arguments produisent le même résultat
- $\Rightarrow$ L'évaluation de la fonction n'a aucun «effet de bord»

Des fonctions, oui, mais des «vraies» fonctions

Des fonctions. . . au sens «mathématique» du terme !

Fonction pure

Le résultat de l'appel à une fonction dépend uniquement des arguments fournis

- ⇒ Les mêmes arguments produisent le même résultat
- ⇒ L'évaluation de la fonction n'a aucun «effet de bord»

JavaScript — Contre-exemple illustrant l'absence de «transparence référentielle» !

```
> foo(11) == foo(11)
false
```

Langages fonctionnels purs vs. impurs

Langage fonctionnel pur

1. Fonctions = «citoyens» de première classe
   Programme = fonctions et composition de fonctions
   $\Rightarrow$ Évaluation d'expressions
   $\neq$ Exécution d'instructions

Langages fonctionnels purs vs. impurs

Langage fonctionnel pur

1. Fonctions = «citoyens» de première classe
   Programme = fonctions et composition de fonctions
   $\Rightarrow$ Évaluation d'expressions
   $\neq$ Exécution d'instructions

2. Manipule des valeurs «immuables»

   > *«[V]alues are abstractions, and hence atemporal, unchangeable and non-instantiated.»*
   >
   > *B.J. MacLennan (1982)*

   $\Rightarrow$ Pas d'effet de bord

Dans un langage fonctionnel pur, la notion de «variable» n'existe pas

SAL = Single-Assignment Languages

On peut avoir des définitions, mais pas des affectations

Définition

Introduit une association entre un identificateur et une valeur

Affectation

Modifie la valeur associée à un identificateur

Dans un langage fonctionnel pur, la notion de «variable» n'existe pas

Exemple Elm — `elm repl`

```
> x = 10
10 : number

> x = x + 1
```

Dans un langage fonctionnel pur, la notion de «variable» n'existe pas

Exemple Elm — `elm repl`

```
> x = 10
10 : number

> x = x + 1


-- BAD RECURSION ----- [...]
`x` is defined directly in terms of itself, causing an
    infinite loop.

5| x = x + 1
   ^
Maybe you are trying to mutate a variable?
Elm does not have mutation, so when I see `x` defined
in terms of `x`, I treat it as a recursive definition.
Try giving the new value a new name[...]
```

Une définition est comme une équation pour laquelle le langage fournit une solution… si elle existe

Exemple Elm — `elm repl`

```
> fac n = if n == 0 then 1 else n * fac (n - 1)
<function> : number -> number1


> fac 5
120 : number

> fac 10
3628800 : number

> fac -1
RangeError: Maximum call stack size exceeded
```

Une définition est comme une équation pour laquelle le langage fournit une solution. . . si elle existe

Une définition est comme une équation pour laquelle le langage fournit une solution. . . si elle existe

Miranda

```
% cat rec.m
x = x + 1

y = y + 1
    where
      y = 2 || Defn. locale utilisee par y+1

% mira rec.m
Miranda x

BLACK HOLE

Miranda y
3
```

Une définition est comme une équation pour laquelle le langage fournit une solution. . . si elle existe

Miranda

```
% cat rec2.m
x = 1 + x
y = 1 : y

% mira rec2.m
Miranda x

BLACK HOLE

Miranda y
```

Une définition est comme une équation pour laquelle le langage fournit une solution... si elle existe

Miranda

```
% cat rec2.m
x = 1 + x
y = 1 : y

% mira rec2.m
Miranda x

BLACK HOLE

Miranda y

[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
   ,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,1,1,C-c C-c<<...interrupt>>
```

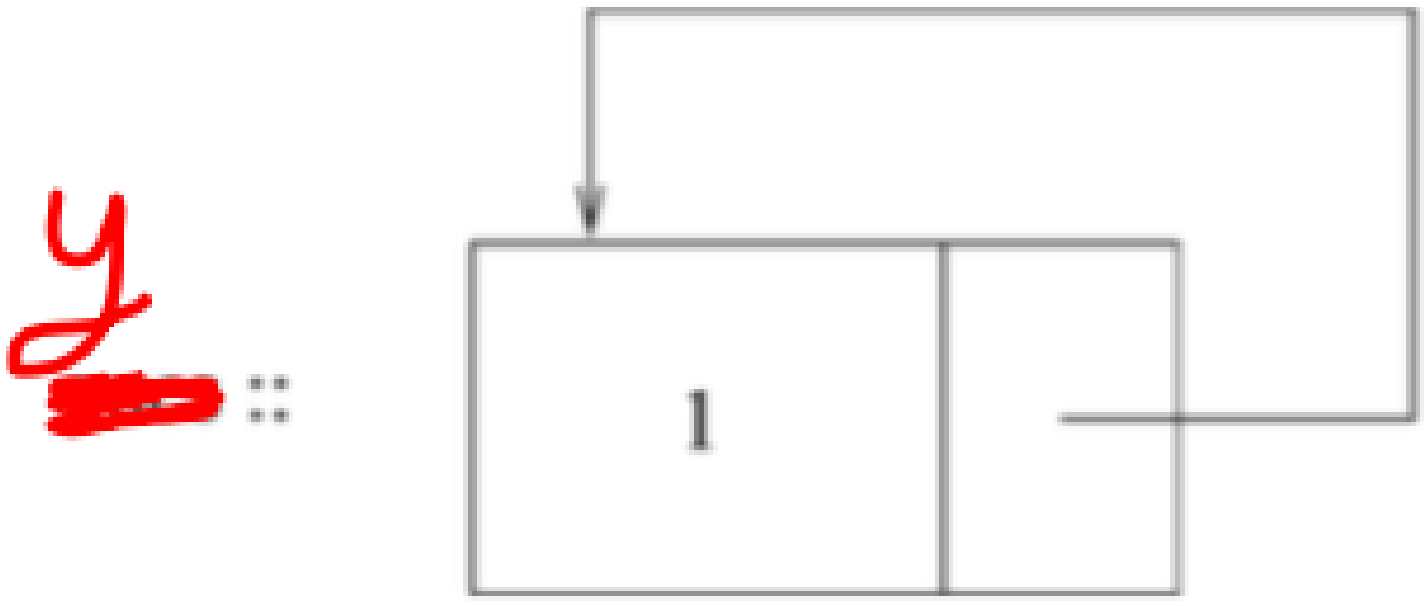


Fig. 2. An infinite list of 1's.

Langages fonctionnels purs vs. impurs

Langage fonctionnel impur

1. Fonctions = «citoyens» de première classe
   Programme = fonctions et composition de fonctions
   $\Rightarrow$ Évaluation d'expressions

Langages fonctionnels purs vs. impurs

Langage fonctionnel impur

1. Fonctions = «citoyens» de première classe
   Programme = fonctions et composition de fonctions
   $\Rightarrow$ Évaluation d'expressions

2. Manipule surtout — mais pas exclusivement — des valeurs «immuables»

$\approx$ Les effets de bord sont possibles mais de façon restreinte/limitée

Les avantages de l'immutabilité

En général :

> **Immutability** *is at the core of functional programming. Immutability is the idea that once a value is declared, it is unchangeable and thus* **makes behaviours** *within your programme* **far more predictable**.
>
> *B.J. MacLennan (1982)*

Les avantages de l'immutabilité

En général :

> **Immutability** *is at the core of functional programming. Immutability is the idea that once a value is declared, it is unchangeable and thus* **makes behaviours** *within your programme* **far more predictable**.
>
> *B.J. MacLennan (1982)*

Dans le contexte de la programmation concurrente :

> **Immutability** *eliminates a whole array of possible issues in a program, ranging from race conditions caused by concurrent code, to uncontrolled global state modification.*

Mais «l'immutabilité» totale est impossible si on veut interagir avec «le monde réel»

[O]bjects correspond to real world entities, and hence exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared.

B.J. MacLennan (1982)

Mais «l'immutabilité» totale est impossible si on veut interagir avec «le monde réel»

[O]bjects correspond to real world entities, and hence exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared.

B.J. MacLennan (1982)

Mais «l'immutabilité» totale est impossible si on veut interagir avec «le monde réel»

[O]bjects correspond to real world entities, and hence exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared.

B.J. MacLennan (1982)

Haskell	`IO` *Monad*, `State` *Monad*, etc.
Elm	Échange de messages entre l'application (fonct. pure) et l'environnement
Elixir	Échange de messages entre processus (acteurs fonct. purs !)

Certains langages sont considérés fonctionnels, même si impurs, car ils restreignent les effets de bord⋆

Clojure

```
user=> (def x (ref 0))
#'user/x
user=> (deref x)
0

user=> (alter x (fn [n] (+ n 2)))
Execution error (IllegalStateException) at [...]
No transaction running

user=> (dosync (alter x (fn [n] (+ n 2))))
2

user=> (deref x)
2
```

Aperçu

2.1 Expressions lambdas

Expression lambda (λ-*expression*) = Expression qui produit une valeur qui est une fonction, mais anonyme

Elm

```
-- Une valeur de type fonction
> \x -> 2 * x
<function> : number -> number

-- Un appel de la fonction
> (\x -> 2 * x) 8
16 : number

-- Une association pour l'identificateur f
> f = \x -> 2 * x
<function> : number -> number

-- Un appel a la fonction associee a f
> f 8
16 : number
```

Les expressions lambda ont initialement été introduites en Lisp (1958)

Lisp

```
;; Une valeur:  \x -> 2 * x + 1
(lambda (x) (+ (* 2 x) 1))

;; Un appel
((lambda (x) (+ (* 2 x) 1)) 8)   ;; => 17
```

Syntaxe Lisp

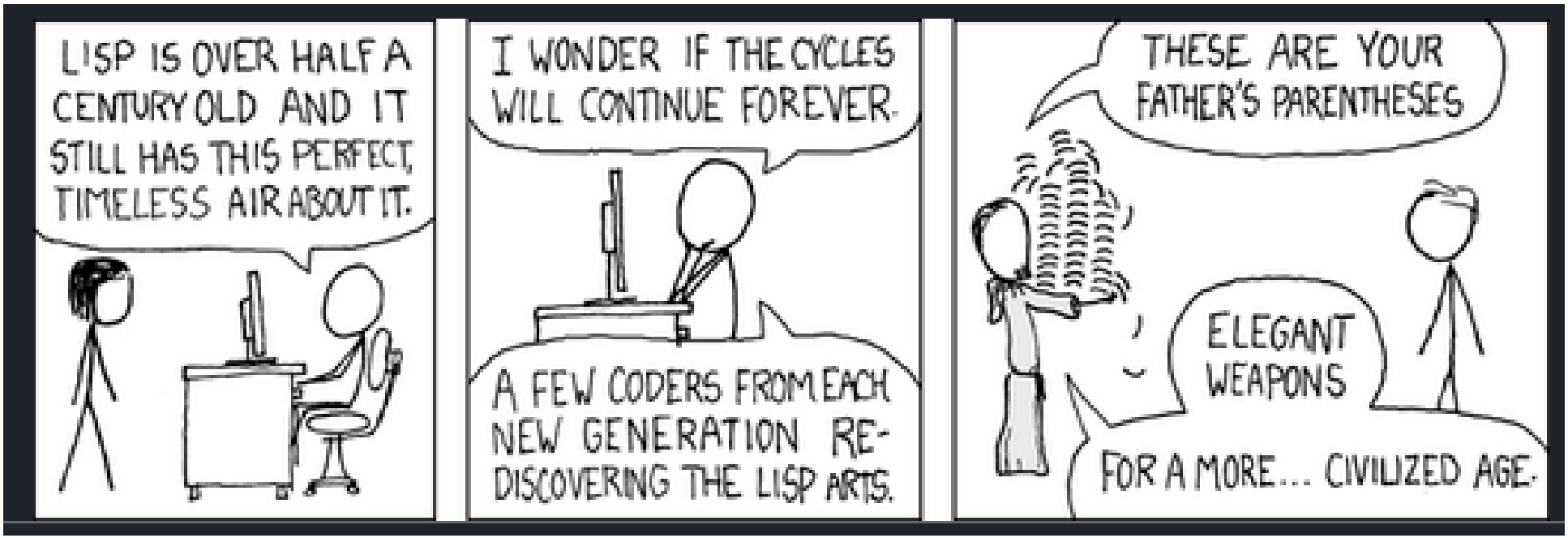
$$\mathit{expr} ::= \mathit{nombre} \mid \mathit{atome} \mid (e_1\ e_2 \ldots\ e_n)$$

Les expressions lambda ont initialement été introduites en Lisp (1958)

Lisp

```
;; Une valeur:  \x -> 2 * x + 1
(lambda (x) (+ (* 2 x) 1))

;; Un appel
((lambda (x) (+ (* 2 x) 1)) 8)   ;; => 17
```



Les expression lambdas sont maintenant présentes dans de nombreux langages... fonctionnels ou non

Un appel avec l'argument 8 à une «valeur» (fonction) qui reçoit un nombre et retourne le double de ce nombre plus 1

```
JavaScript:
(function(x) { return 2 * x + 1; })(8)


C++11 (2011):
[](int x) { return 2 * x + 1; }( 8 )


Java 8 (2014):
((Function<Integer,Integer>)
    (x -> 2 * x + 1)).apply(8)
```

Miranda, un langage purement fonctionnel, n'a pas d'expressions lambda

Miranda = un des langages qui a le plus influencé Haskell = Langage fonctionnel pur et paresseux !

Miranda n'a que des fonctions nommées explicitement, ou des «sections»

```
Miranda:
res = map double [1, 3, 7]
        where
          double x = 2 * x


res2 = map (2 *) [1, 3, 7]
```

Les expressions lambdas en Java 8 permettent de diminuer le code *boilerplate*, par ex., pour les *threads*

Exemple : Interface et classe pour les *threads*

Java : Interface `Runnable` et classe `Thread`

```
interface Runnable {
    void run()
}


class Thread implements Runnable {
    Thread() { ... }
    Thread( Runnable target ) { ... }
    Thread( ... ) { ... }
    ...
}
```

Les expressions lambdas en Java 8 permettent de diminuer le code *boilerplate*, par ex., pour les *threads*

On veut créer un *thread* pour un appel à la méthode `foo`.

Java avant Java 8 : Classe interne anonyme avec méthode `run`

```
new Thread( new Runnable() {
    public void run() {
        foo( ... );
    }
}
```

Les expressions lambdas en Java 8 permettent de diminuer le code *boilerplate*, par ex., pour les *threads*

On veut créer un *thread* pour un appel à la méthode `foo`.

Java avant Java 8 : Classe interne anonyme avec méthode `run`

```
new Thread( new Runnable() {
    public void run() {
        foo( ... );
    }
}
```

Java depuis Java 8 : Expression lambda

```
new Thread( () -> foo( ... ) );
```

2.2 Inférence de types

Typage statique vs. Typage dynamique

Typage statique

= On vérifie les types à la compilation
⇒ Aucune erreur de type ne survient durant l'exécution

Typage dynamique

= On vérifie les types durant l'exécution
⇒ Une erreur de type peut survenir durant l'exécution

Typiquement, dans les langages impératifs, le typage statique est explicite

Typage statique explicite

= Les types doivent être spécifiés explicitement par le programmeur !

C

```
% cat exemples.cpp
...
int f( int x ) {
    return x + "a";
}
...
% gcc exemples.cpp
exemples.cpp:5:14: warning: adding 'int'
      to a string does not append to the string
      [-Wstring-plus-int]
    return x + "a";
               ~~^~~~
```

Dans les langages avec typage dynamique, les types n'ont pas besoin d'être spécifiés

Les types sont implicites

Ruby

```
>> def f( x, y )
     if x != 0 then x * y else x + "a" end
   end
=> :f


>> f( 8, 2 )
```

Dans les langages avec typage dynamique, les types n'ont pas besoin d'être spécifiés

Les types sont implicites

Ruby

```
>> def f( x, y )
     if x != 0 then x * y else x + "a" end
   end
=> :f


>> f( 8, 2 )
=> 16

>> f( 0, 2 )
```

Dans les langages avec typage dynamique, les types n'ont pas besoin d'être spécifiés

Les types sont implicites

Ruby

```
>> def f( x, y )
     if x != 0 then x * y else x + "a" end
   end
=> :f


>> f( 8, 2 )
=> 16

>> f( 0, 2 )
err.rb:6:in `+':
   String can't be coerced into Fixnum (TypeError)
   [...]
```

Le typage statique peut aussi être implicite $\Rightarrow$ Vérification à la compilation, sans spécification de type

Introduit en ML (Milner, 1973)

Le typage statique peut aussi être implicite ⇒ Vérification à la compilation, sans spécification de type

Introduit en ML (Milner, 1973)

Elm

```
> f1 x y z = if x then y + 1 else z
<function> : Bool -> number -> number -> number

> f2 x y = if x == [] then y else x ++ y
<function> : List a -> List a -> List a
```

Le typage statique peut aussi être implicite ⇒ Vérification à la compilation, sans spécification de type

Introduit en ML (Milner, 1973)

Elm

```
> f1 x y z = if x then y + 1 else z
<function> : Bool -> number -> number -> number

> f2 x y = if x == [] then y else x ++ y
<function> : List a -> List a -> List a



> compose f g = \x -> g (f x)
<function> : (a -> b) -> (b -> c) -> a -> c

> compose ((+) 10) toString 3
"13" : String
```

Le typage statique peut aussi être implicite ⇒ Vérification à la compilation, sans spécification de type

Elm

```
f5 x y =
  if x /= 0 then
    x / y
  else
    x + "a"

-- TYPE MISMATCH -- src/erreurs-compilation.elm
The right side of (+) is causing a type mismatch.
103|     x + "a"
             ^^^

(+) is expecting the right side to be a:
    number
But the right side is:
    String

Hint: To append strings, you need to use (++), not (+).
```

Typage statique implicite : Elm vs. Haskell

Les messages d'erreur du compilateur Elm sont généralement très compréhensibles !

Elm

```
plus1 : Int -> Int
plus1 x = x + 1

r = plus1 2.0

-- TYPE MISMATCH -- src/erreurs-compilaton.elm
The argument to function `plus1` is causing a mismatch.
6|       plus1 2.0
                ^^^
Function `plus1` is expecting the argument to be:
   Int
But it is:
   Float

Hint: Elm does not automatically convert between Ints
    and Floats.
Use `toFloat` and `round` to do specific conversions.
```

Typage statique implicite : Elm vs. Haskell

Les messages d'erreur du compilateur Elm sont généralement très compréhensibles !

Haskell

```
plus1 :: Int -> Int
plus1 x = x + 1

res = plus1 2.0

exemples.hs:8:13:
    No instance for (Fractional Int) arising from the
    literal '2.0'
    In the first argument of 'plus1', namely '2.0'
    In the expression: plus1 2.0
    In an equation for 'res': res = plus1 2.0
```

Typage statique implicite : Elm vs. Haskell

Les messages d'erreur du compilateur Elm sont généralement très compréhensibles !

Elm

```
map_ f ls =
  case ls of
    [] -> []
    x :: rest -> f x :: map_ f rest

res = map_ list ((+) 1)

-- TYPE MISMATCH -- src/erreurs-compilation.elm
The 2nd argument to function `map_` is causing a
    mismatch.
42|       map_ list ((+) 1)
                    ^^^^^
Function `map_` is expecting the 2nd argument to be:
    List a
But it is:
    number -> number

Hint: It looks like a function needs 1 more argument.
```

Typage statique implicite : Elm vs. Haskell

Les messages d'erreur du compilateur Elm sont généralement très compréhensibles !

Haskell

```
map_ f ls =
  case ls of
    [] -> []
    x : rest -> f x : map_ f rest

res = map_ list ((+) 1)

exemples.hs:14:13:
 Couldn't match expected type 't0 -> t' with actual
    type '[Integer]'
 Relevant bindings include res :: [t] ([...])
 In the first argument of 'map_', namely 'list'
 In the expression: map_ list ((+) 1)

exemples.hs:14:19:
 Couldn't match expected type '[t0]' with actual type '
    Integer -> Integer'
 Probable cause: '(+)' is applied to too few arguments
```

Petite «parenthèse» concernant le «typage» (sic) dynamique... en JavaScript

Question⋆ : Quelle est la valeur de chacune des expressions suivantes ?

JavaScript

```
$ node
> 9 + "3"      // a:
??

> 9 - "3"      // b:
??

> "9" * 3      // c:
??

> "9" == 9     // d:
??
```

Petite «parenthèse» concernant le «typage» (sic) dynamique... en JavaScript

Question⋆ : Quelle est la valeur de chacune des expressions suivantes ?

JavaScript : Réponse 2

```
$ node
> 9 + "3"      // a:
'93'

> 9 - "3"      // b:
6

> "9" * 3      // c:
27

> "9" == 9     // d:
true
```

Petite «parenthèse» concernant le «typage» (sic) dynamique... en JavaScript

JavaScript

```
> +[]
??

> [] + []
??

> true+true
??

> +!![]
??

> (![]+[])[+!![]]
??
```

Source: http://jazcash.com/a-javascript-journey-with-only-six-characters/

Petite «parenthèse» concernant le «typage» (sic) dynamique... en JavaScript

JavaScript

```
> +[]
0

> [] + []
""

> true+true
2

> +!![]
1

> (![]+[])[+!![]]
'a'
```

Source: http://jazcash.com/a-javascript-journey-with-only-six-characters/

«*JavaScript : The Complete Reference*» «*JavaScript : The Good Parts*»

ML a aussi été le premier langage à introduire les types et fonctions génériques

Polymorphisme paramétrique

«*a function or a data type can be written generically so that it can handle values identically without depending on their type.*»

Wikipedia

ML a aussi été le premier langage à introduire les types et fonctions génériques

Polymorphisme paramétrique

«*a function or a data type can be written generically so that it can handle values identically without depending on their type.*»

Wikipedia

Elm

```
> longueur liste =
    case liste of
       [] -> 0
       _ :: rest -> 1 + longueur rest
<function> : List a -> number

> longueur [10, 20]
2 : number

> longueur [ ["ab", "de"], [], ["x"] ]
3 : number
```

Typage implicite et types génériques sont maintenant disponibles dans divers langages impératifs

Inférence de types

C++11

```
int (*f1)(int) = [](int x) { return 2 * x; };
auto f2 = f1;

std::cout << f1(8) << "\n";
std::cout << f2(8) << "\n";
```

Typage implicite et types génériques sont maintenant disponibles dans divers langages impératifs, dont Java

Java 5 (2004) et Java 7 (2011) : Types génériques

```
public class ArrayList<E> extends AbstractList<E>
                          implements List<E> {
  ...
}


// Java 5:
List<String> list1 = new ArrayList<String>();



// Java 7:
List<String> list1 = new ArrayList<>();
```

Typage implicite et types génériques sont maintenant disponibles dans divers langages impératifs, dont Java

Java 8 (2014) : Paramètres des expr. lambdas

```
Function<Integer,Integer> f1 =
    (Integer x) -> 2 * x;

Function<Integer,Integer> f1 =
    x -> 2 * x;
```

Typage implicite et types génériques sont maintenant disponibles dans divers langages impératifs, dont Java

Inférence de types

Java 8 (2014) : Paramètres des expr. lambdas

```
Function<Integer,Integer> f1 =
    (Integer x) -> 2 * x;

Function<Integer,Integer> f1 =
    x -> 2 * x;
```

Java 10 (2018) : `var` pour variables locales

```
Function<Integer,Integer> f1 = x -> 2 * x;
var f2 = f1;
```

Typage implicite et types génériques sont maintenant disponibles dans divers langages impératifs, dont Java

Inférence de types

Java 8 (2014) : Paramètres des expr. lambdas

```
Function<Integer,Integer> f1 =
    (Integer x) -> 2 * x;

Function<Integer,Integer> f1 =
    x -> 2 * x;
```

Java 10 (2018) : `var` pour variables locales

```
Function<Integer,Integer> f1 = x -> 2 * x;
var f2 = f1;
```

Java 11 (2018) : `var` pour paramètres ($\Rightarrow$ annotations)

```
Function<Integer,Integer> f1 =
    x -> 2 * x;

Function<Integer,Integer> f1 =
    (@Nonnull var x) -> 2 * x;
```

2.3 Traitement de collections avec `map`, `reduce` *et al.*

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean, Sanjay Ghemawat

OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150


hadoop


APACHE
kafka
A distributed streaming platform


APACHE
STORM™


APACHE
Spark™

Approche Map/Reduce ⇒ Traiter des collections de données sans boucle — ou sans récursion

`Map`

«Transforme» chacun des éléments d'une collection en un nouvel élément pour produire une nouvelle collection (possiblement d'un type différent)

`Reduce`

Combine les éléments d'une collection en une *unique* valeur à l'aide d'un opérateur binaire (associatif, parfois aussi commutatif)

Approche introduite en Lisp (1958) via des fonctions d'ordre supérieur et expr. lambdas

Lisp

```
(mapcar #'(lambda(x) (* x 2)) '(1 2 3 4 5))
=> (2 4 6 8 10)


(reduce #'+ '(1 2 3 4 5))
=> 15
```

Permet souvent d'exprimer des calculs complexes de façon succincte et modulaire

Exemple : Somme de la valeur absolue des éléments impairs d'une collection `a`

Ruby — Style itératif et impératif

```ruby
def somme_abs_impairs( a )
  somme = 0
  for n in a
    somme += n.abs if n.odd?
  end

  return somme
end
```

Permet souvent d'exprimer des calculs complexes de façon succincte et modulaire

Exemple : Somme de la valeur absolue des éléments impairs d'une collection `a`

Ruby — Style récursif… et fonctionnel !

```ruby
def somme_abs_impairs( a )
  return 0 if a == []

  if a.first.odd?
    a.first.abs + somme_abs_impairs(a.drop(1))
  else
    somme_abs_impairs(a.drop(1))
  end
end
```

Permet souvent d'exprimer des calculs complexes de façon succincte et et modulaire

Exemple : Somme de la valeur absolue des éléments impairs d'une collection `a`

Ruby — Style fonctionnel via `Enumerable`

```ruby
def somme_abs_impairs( a )
  a.filter { |n| n.odd? }
   .map { |n| n.abs }
   .reduce(:+)
end
```

Depuis Java 8, une approche semblable est possible, avec les `Stream`s

Exemple : Somme de la valeur absolue des éléments impairs

Java — Style itératif et impératif

```java
int somme = 0;
for ( int x: a ) {
  if ( x % 2 != 0 ) {
    somme += Math.abs(x);
  }
}
```

Depuis Java 8, une approche semblable est possible, avec les `Streams`

Exemple : Somme de la valeur absolue des éléments impairs

Java — Style itératif et impératif

```
int somme = 0;
for ( int x: a ) {
  if ( x % 2 != 0 ) {
    somme += Math.abs(x);
  }
}
```

Java — Style fonctionnel (avec `Streams`)

```
int somme = Arrays.stream( a )

                  .filter( x -> x % 2 != 0 )
                  .map( Math::abs )
                  .sum();
```

Depuis Java 8, une approche semblable est possible, avec les `Streams`

Exemple : Somme de la valeur absolue des éléments impairs

Java — Style itératif et impératif

```
int somme = 0;
for ( int x: a ) {
  if ( x % 2 != 0 ) {
    somme += Math.abs(x);
  }
}
```

Java — Style fonctionnel (avec `Streams`) et **parallèle !**

```
int somme = Arrays.stream( a )
                  .parallel()
                  .filter( x -> x % 2 != 0 )
                  .map( Math::abs )
                  .sum();
```

Un exemple canonique en traitement de données : `WordCount`

Le problème

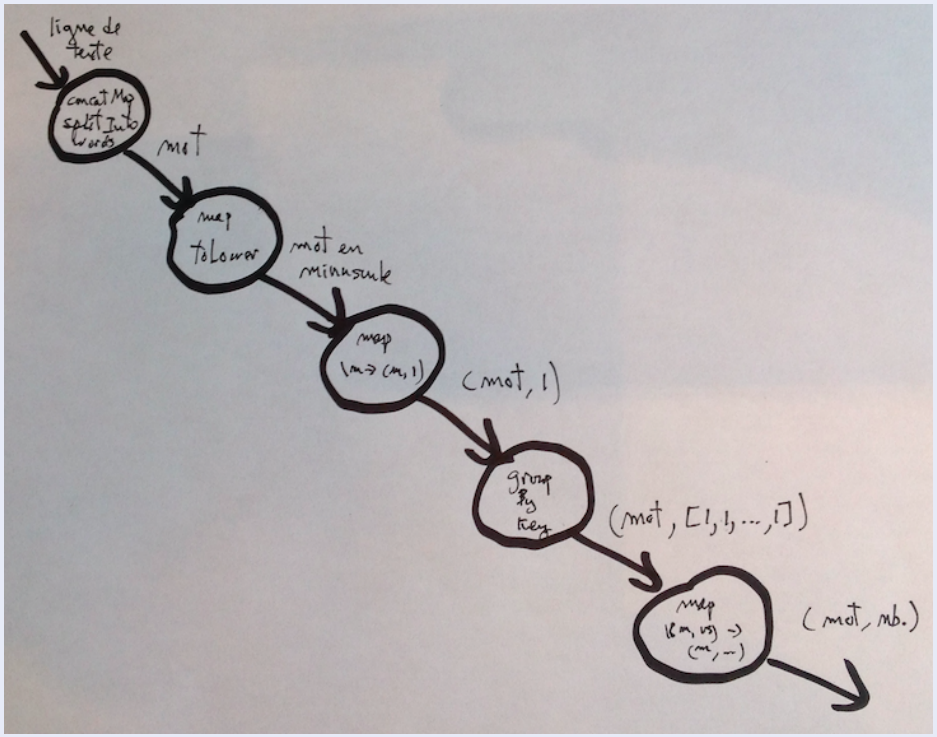
On veut compter le nombre d'occurrences de chacun des mots (suite de lettres) d'un texte (suite de lignes).

Un exemple canonique en traitement de données : `WordCount`

Le problème

On veut compter le nombre d'occurrences de chacun des mots (suite de lettres) d'un texte (suite de lignes).

Une solution de style «traitement de flux de données»



Exemple d'entrée

```
Vous deux vous allez etre sages, cette annee!
Si jamais je recois un hibou qui me dit que vous
avez fait exploser les toilettes...

- Faire exploser les toilettes? On n'a jamais fait ça

- Mais c'est une bonne idee. Merci maman!
```

Source : `https:`

`//booknode.com/harry_potter_tome_1_harry_potter_a_l_ecole_des_sorciers_0983/extraits`

Sortie produite (les mots sont tous en minuscules)

```
je => 1
deux => 1
sages => 1
hibou => 1
etre => 1
un => 1
mais => 1
...
```

```
jamais => 2
une => 1
recois => 1
vous => 3
que => 1
idee => 1
qui => 1
...
```

```
les => 2
toilettes => 2
maman => 1
on => 1
n => 1
a => 2
c => 1
...
```

Une mise en œuvre de `WordCount` en Elm

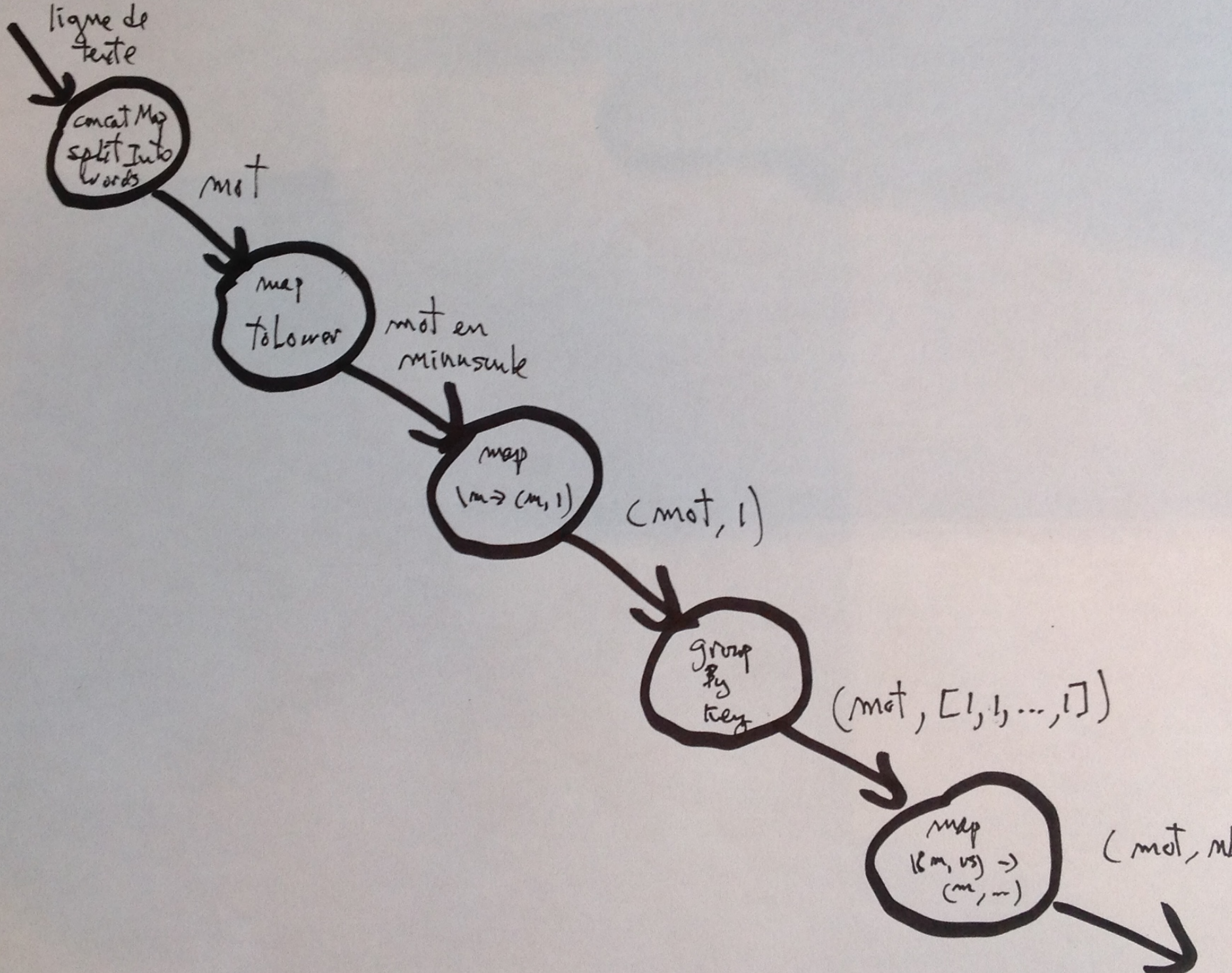
Elm

```
lines
      --
  |> concatMap splitIntoWords
      --
  |> map String.toLower
      --
  |> map (\m -> ( m, 1 ))
      --
      --
  |> groupByKey -- ≈ Shuffle de MapReduce
      --
  |> map (\( k, vs ) -> ( k, length vs ))
      --
```

Une mise en œuvre de `WordCount` en Elm

Elm

```
lines
      -- ["abc def,   DEF ","== abc AbC?"]
  |> concatMap splitIntoWords
      -- ["abc","def","DEF","abc","AbC"]
  |> map String.toLower
      -- ["abc","def","def","abc","abc"]
  |> map (\m -> ( m, 1 ))
      -- [("abc",1),("def",1),("def",1),
      --  ("abc",1),("abc",1)]
  |> groupByKey  -- Shuffle
      -- [("abc",[1,1,1]),("def",[1,1])]
  |> map (\( k, vs ) -> ( k, length vs ))
      -- [("abc",3),("def",2)]
```


ligne de texte
concatMap splitIntoWords
mot
map toLower
mot en minuscule
map (m -> (m, 1))
(mot, 1)
group By key
(mot, [1, 1, ..., 1])
map ((m, vs) -> (m, ...)
(mot, m

Des programmes de ce style peuvent être vus comme des instances de l'approche Unix

= Diviser-pour-régner... non récursif !

La philosophie Unix selon McIlroy et al.

> *(i) Make each program do one thing well. [...]*
> *(ii) Expect the output of every program to become the input to another, as yet unknown, program. [...]*
>
> «*Unix Time-Sharing System Forward*» (1978)

Un script `bash` pour `WordCount`

```
$ cat word-count.sh
#!
tr -cs "[:alpha:]" "\n" |
  tr "[:upper:]" "[:lower:]" |
  sort |
  uniq -c |
  awk '{print $2 " => " $1}'
```

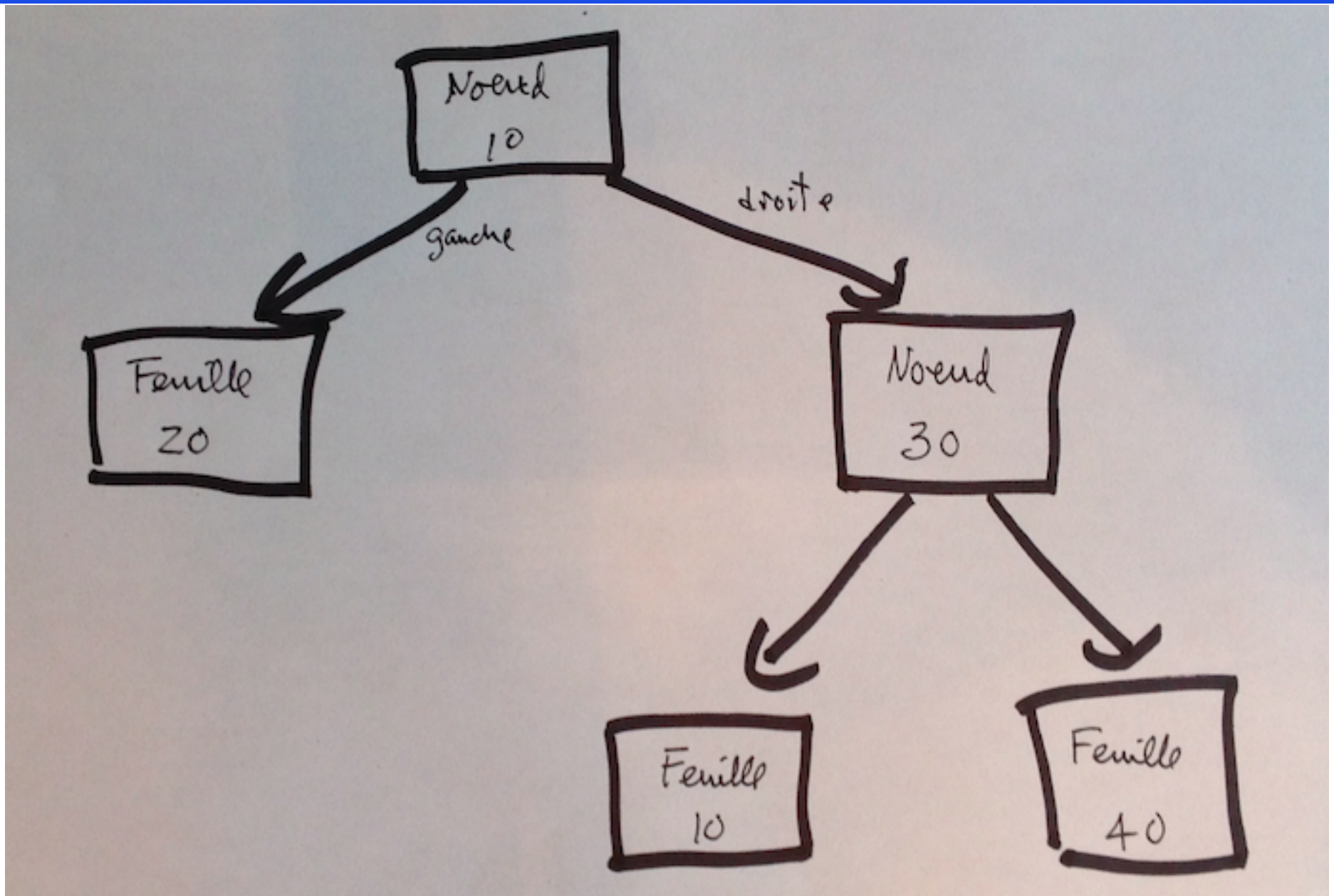
Et voici une mise en œuvre de `WordCount` en Java 8

Java

```
lines
   .flatMap( WordCount::splitInWords )
   .map( String::toLowerCase )
   .map( word -> new SimpleEntry<>( word, 1 ) )
   .collect(
     Collectors.groupingBy(Map.Entry::getKey) )
   .entrySet().stream()
   .map( e ->
         new SimpleEntry<>(e.getKey(),
                           e.getValue().size()) )
   .collect( Collectors.toList() );
```

2.4 Types algébriques de données et «*pattern-matching*»

Soit des arbres binaires pour lesquels on veut calculer la `somme` des attributs `valeur`



Java : Solution avec classe abstraite et sous-classes

```
public abstract class Arbre {
    protected Integer valeur;

    abstract Integer somme();
}

class Feuille extends Arbre {
    Feuille( Integer valeur ) { ... }

    Integer somme() { return valeur; }
}

class Noeud extends Arbre {
    Arbre gauche, droite;
    Noeud( ... ) { ... }

    Integer somme() {
       return valeur + gauche.somme() + droite.somme();
    }
}
```

Elm : Solution avec type algébrique de données (générique) et *pattern-matching* via expression `case`

```
type Arbre t
   = Feuille t
   | Noeud t (Arbre t) (Arbre t)


somme : Arbre number -> number
somme arbre =
  case arbre of
    Feuille v ->
      v

    Noeud v gauche droite ->
      v + (somme gauche) + (somme droite)
```

«*A pattern is [a] way of characterizing a group of objects and binding local names to objects that match some property in the classification.*»

Type produit vs. Type somme

Type produit = Produit cartésien de types

Un nouveau type est composé en combinant divers éléments, possiblement de types différents.

Exemples :

- `struct` : C, C++
- `class` et attributs : Java, C++
- `record` : Elm
- Etc.

Type produit vs. Type somme

Type somme = Somme disjointe de types

Un nouveau type est composé en sélectionnant un élément parmi diverses possibilités.

Exemples :

- `union` : C, C++
- Groupe de sous-classes d'une classe abstraite : Java, C++
- Etc.

Type algébrique de données = Somme de produits

Donc combine type produit et type somme en un même nouveau type

Elm

```
type UnNouveauType
   = Cons1 T1 ... Tk1  -- T1 × ... × Tk1
   | Cons2 T1 ... Tk2
   | ...
   | Consn T1 ... Tkn  -- T1 × ... × Tkn
```

Type algébrique de données = Somme de produits

Donc combine type produit et type somme en un même nouveau type

Elm

```
type UnNouveauType
  = Cons1 T1 ... Tk1  -- T1 × ... × Tk1
  | Cons2 T1 ... Tk2
  | ...
  | Consn T1 ... Tkn  -- T1 × ... × Tkn
```

Elm : Type algébrique pour `Arbre` binaire générique

```
type Arbre t
   = Feuille t
   | Noeud t (Arbre t) (Arbre t)
```

Les types algébriques peuvent aider à expliciter certaines contraintes sur les données

Exemple : On veut conserver des informations de contact pour des clients et on a trois (3) possibilités :

1. Uniquement une adresse de courriel
2. Uniquement une adresse postale
3. Une adresse de courriel **et** une adresse postale

Donc : Impossible de n'avoir aucune forme d'adresse.

Les types algébriques peuvent aider à expliciter certaines contraintes sur les données

Exemple : On veut conserver des informations de contact pour des clients et on a trois (3) possibilités :

1. Uniquement une adresse de courriel
2. Uniquement une adresse postale
3. Une adresse de courriel **et** une adresse postale

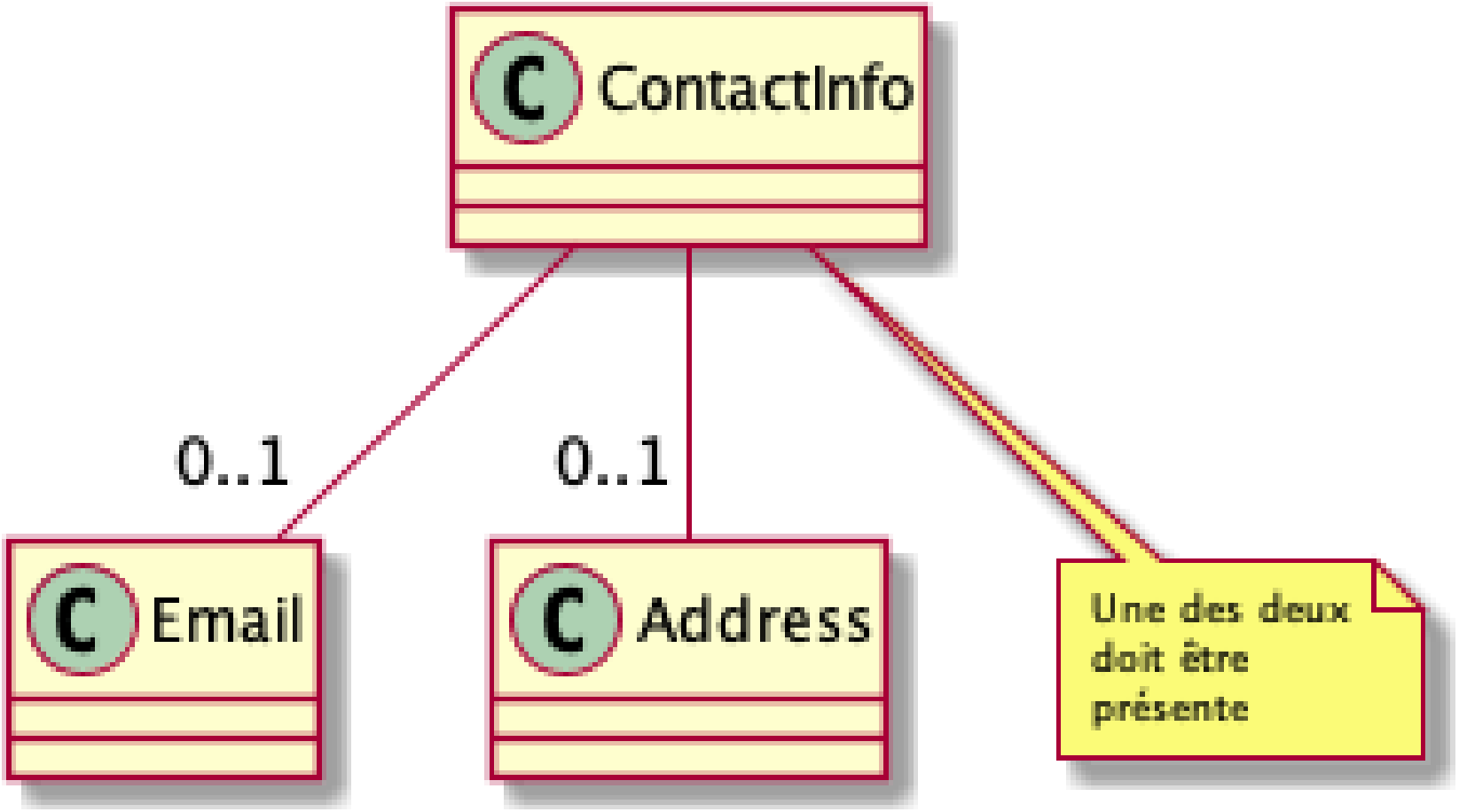
Donc : Impossible de n'avoir aucune forme d'adresse.

- On veut envoyer un message, en priorisant l'envoi par l'intermédiaire de l'adresse de courriel si présente, sinon par l'adresse postale.

Note : Inspiré de `https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/`

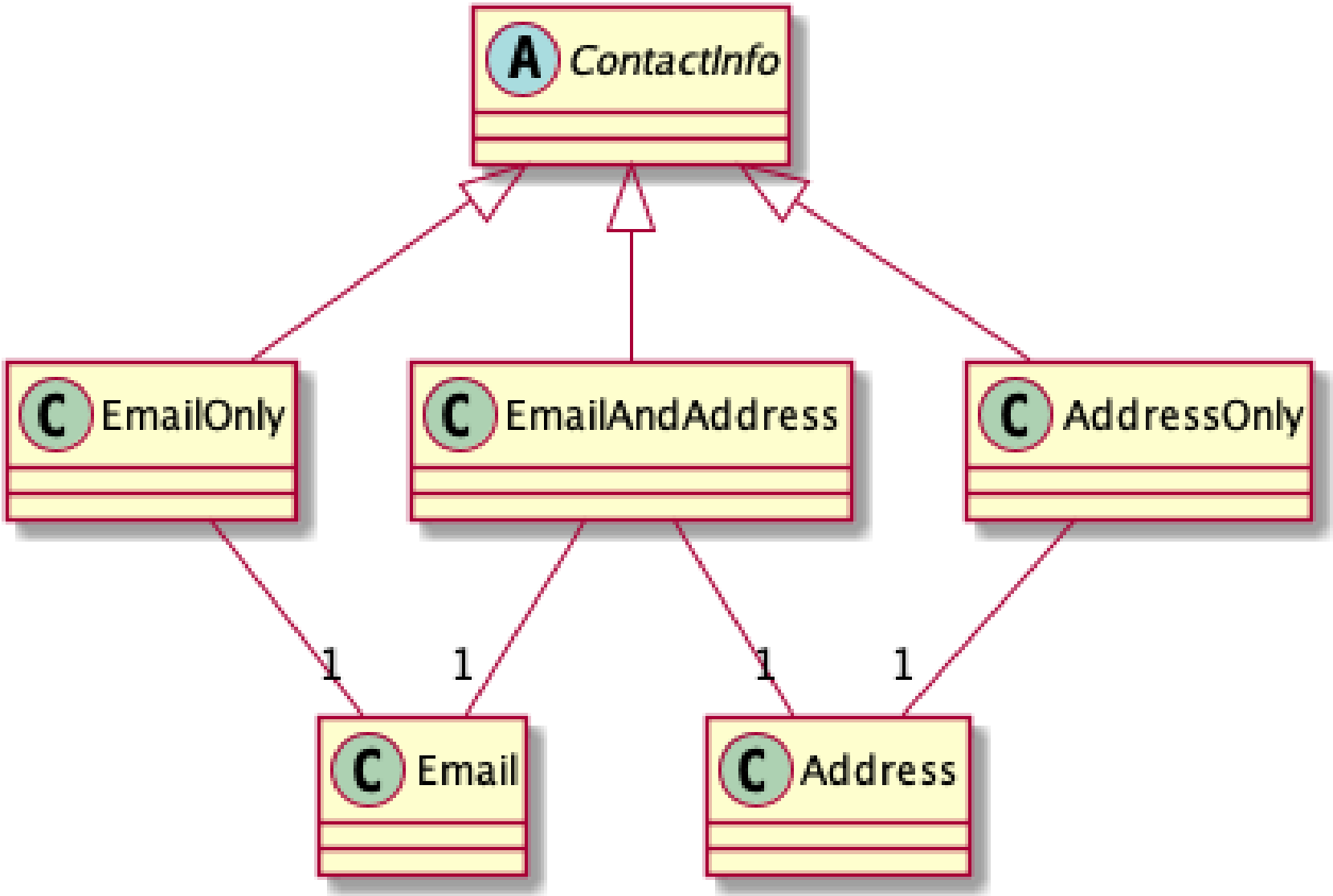
Un modèle UML pour une classe `ContactInfo` avec des champs (liens) possiblements vides

L'obligation d'avoir un des deux champs ne peut être spécifié que de façon informelle

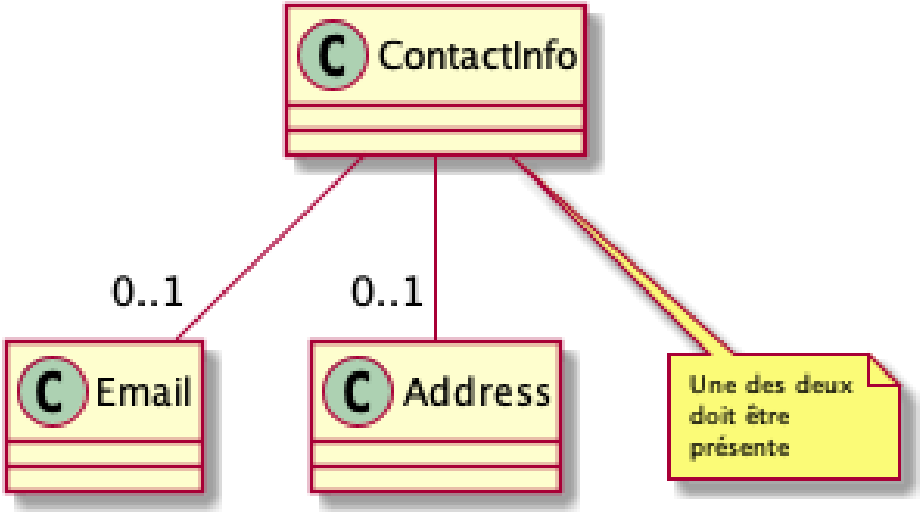


Un modèle UML pour une classe `ContactInfo` avec les trois possibilités explicitement indiquées

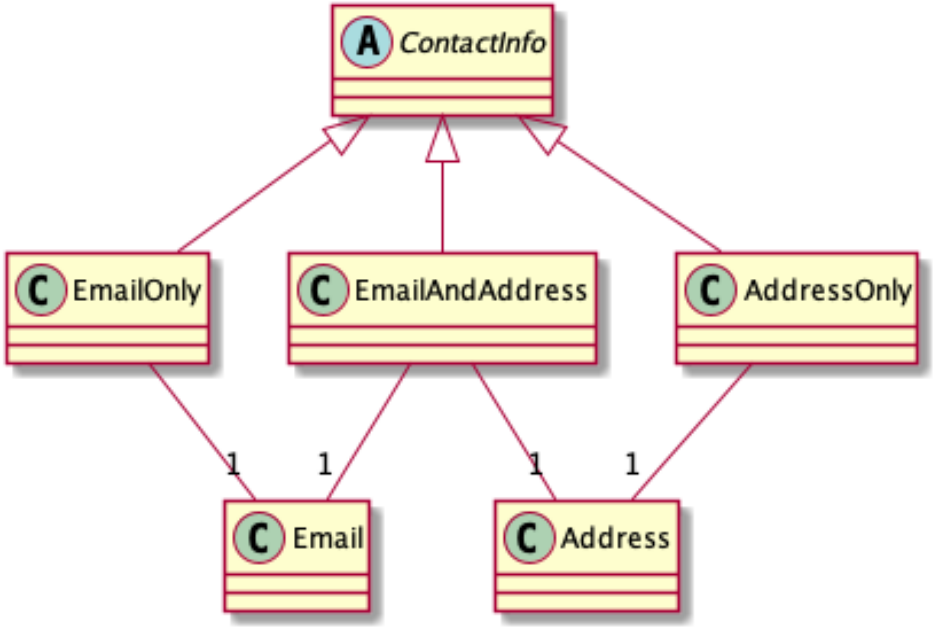
L'obligation d'avoir un des deux champs est claire et explicite



1. Avec des champs optionnels (possiblement `null`)



2. Avec les trois possibilités explicites



Une solution Elm utilisant le premier modèle

Le type `ContactInfo` avec un `record` et des champs optionnels

Elm : Le type `Maybe` indique que le champ peut être présent (`Just v`) ou non (`Nothing`)

```
type alias ContactInfo =
    { email : Maybe Email
    , address : Maybe Address
    }
```

Une solution Elm utilisant le premier modèle

L'envoi d'un message

Elm

```
msgForContact : ContactInfo -> Msg
msgForContact ci =
  case ci.email of
    Just email ->
      msgForEmail email

    Nothing ->
      case ci.address of
        Just addr ->
          msgForAddress addr

        Nothing ->
          ?
```

Une solution Elm utilisant le premier modèle

L'envoi d'un message

Elm

```
msgForContact : ContactInfo -> Msg
msgForContact ci =
  case ci.email of
    Just email ->
      msgForEmail email

    Nothing ->
      case ci.address of
        Just addr ->
          msgForAddress addr

        Nothing ->
          Debug.crash "No contact info..."
```

Une solution Elm utilisant le deuxième modèle

Le type `ContactInfo` avec un type algébrique

Elm

```
type ContactInfo
    = EmailOnly Email
    | AdressOnly Address
    | EmailAndAdress Email Address
```

Une solution Elm utilisant le deuxième modèle

L'envoi d'un message

Elm

```
msgForContact : ContactInfo -> Msg
msgForContact ci =
  case ci of
    EmailOnly email ->
      msgForEmail email

    AddressOnly adr ->
      msgForAddress adr

    EmailAndAddress email _ ->
      msgForEmail email
```

Plusieurs langages récents, non fonctionnels, ont introduit types algébriques et/ou *pattern-matching*

- $\Rightarrow$ Rust : `enum` (*sum types*) + expressions `match`
- $\Rightarrow$ Swift : `enum` (*sum types*) + expressions `switch`
- $\approx$ Ruby 2.7 : *pattern-matching* + `case`

Plusieurs langages récents, non fonctionnels, ont introduit types algébriques et/ou *pattern-matching*

- ≈ Java 14 : *Pattern-matching* via `instanceof` et *type test pattern*

Java avant Java 14

```
if (animal instanceof Cat) {
    Cat cat = (Cat) animal;
    cat.meow();
} else if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.woof();
}
```

Plusieurs langages récents, non fonctionnels, ont introduit types algébriques et/ou *pattern-matching*

- ≈ Java 14 : *Pattern-matching* via `instanceof` et *type test pattern*

Java avant Java 14

```
if (animal instanceof Cat) {
    Cat cat = (Cat) animal;
    cat.meow();
} else if (animal instanceof Dog) {
    Dog dog = (Dog) animal;
    dog.woof();
}
```

Java 14 (JEP 305) : *Pattern Matching for* `instanceof` *(Preview)*

```
if (animal instanceof Cat cat) {
    cat.meow();
} else if (animal instanceof Dog dog) {
    dog.woof();
}
```

2.5 Autres

Structure de bloc et indentation de code

Dans certains langages, la structure du code se réflète dans l'indentation du texte

«*Miranda : A Non-Strict Functional Language*», Turner (1985)

Miranda

```
% cat f1.m
f x = x + y
        where
    y = x^2
% mira f1.m
[...]
syntax error - unexpected token "OFFSIDE"
[...]

% cat f2.m
f x = x + y
        where
          y = x^2
% mira f2.m
[...]
Miranda f 9
90
```

Dans certains langages, la structure du code se réflète dans l'indentation du texte

«*Miranda : A Non-Strict Functional Language*», Turner (1985)

Miranda, avec la notation `where`, permet une notation semblable à la notation mathématique

```
racines :: num -> num -> num -> [num]
racines 0 b c
   = []

racines a b c
   = [], if disc < 0
   = [-b / (2 * a)], if disc = 0
   = [(-b - (sqrt disc)) / (2 * a),
      (-b + (sqrt disc)) / (2 * a)], otherwise
         where
            disc = b * b - 4 * a * c
```

Le langage Elm, pur mais strict, lui aussi utilise l'indentation de code pour structurer le code

Elm : Dans un `let`, l'ordre des clauses n'a pas d'importance (idem en Miranda)

```
> let
    x = 2
    y = x + 1
  in 2 * y
6 : number

> let
    y = x + 1
    x = 2
  in 2 * y
6 : number
```

En Miranda et Elm, langages d'expressions (sans *séq. d'inst.*), une mauvaise indentation génère une erreur de syntaxe.

Le langage Python lui aussi utilise l'indentation pour définir la structure des blocs de code

Python

```
def foo( x ):
    if x < 0:
        x = -x
        x *= 2

    return x

print foo( 10 ) # => 10
```

Python

```
def foo( x ):
    if x < 0:
        x = -x
    x *= 2

    return x

print foo( 10 ) # => 20
```

Toutefois, Python est un langage impératif — avec séquences d'instructions — où une mauvaise indentation ne cause pas nécessairement une erreur de syntaxe ☹

Compréhensions de liste

Les compréhensions de liste ressemblent aux ensembles mathématiques définis en compréhension

Exemples d'ensembles mathématiques définis en compréhension

$$
\begin{aligned}
\mathit{Pairs} &= \{n \in \mathcal{N} \mid n \text{ est divisible par } 2\} \\
A \cap B &= \{x \mid x \in A \wedge x \in B\} \\
\{-1, 1\} &= \{n \in \mathcal{Z} \mid |n| = 1\} \\
&\ldots
\end{aligned}
$$

Les compréhensions de liste ressemblent aux ensembles mathématiques définis en compréhension

Exemples d'ensembles mathématiques définis en compréhension

$$
\begin{aligned}
\mathit{Pairs} &= \{n \in \mathcal{N} \mid n \text{ est divisible par } 2\} \\
A \cap B &= \{x \mid x \in A \wedge x \in B\} \\
\{-1, 1\} &= \{n \in \mathcal{Z} \mid |n| = 1\} \\
&\ldots
\end{aligned}
$$

Liste en compréhension (Wikipedia)

[L]istes dont le contenu est défini par filtrage du contenu d'une autre liste selon un principe analogue à celui de la définition en compréhension de la théorie des ensembles.

`https://fr.wikipedia.org/wiki/Liste_en_compr%C3%A9hension`

Les compréhensions de liste ont été introduites en KRC (Turner, 1979), le prédécesseur de Miranda

Miranda : Somme de la valeur absolue des éléments impairs d'une collection `a`

```
|| Avec map, filter et reduce imbriques
reduce (+) (map abs (filter isOdd a))



|| Avec comprehension de liste
reduce (+) [ abs x | x <- a; isOdd x ]
```

$$\Sigma \{ |x| \mid x \in a \wedge isOdd\ x \}$$

Les compréhensions de liste ont ensuite été reprises en Haskell, Python, etc.

Python

```
# Avec map, filter et smallsum imbriques
sum( map( abs, filter( isOdd, a ) ) )


# Avec comprehension de liste
sum( [abs(x) for x in a if isOdd(x)] )
```

Aperçu

Depuis un quart de siècle, on entend beaucoup parler des patrons de conception

«*Design Patterns—Elements of Reusable Obj.-Or. Software*», Gamma *et al.* (1995)

Software Design Pattern

A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. [. . .] Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

`https://en.wikipedia.org/wiki/Software_design_pattern`

Or, les patrons de conception sont souvent liés à des limites du langage utilisé

Deux citations

Norvig [wrote] that 16 of the 23 GOF patterns are «invisible or simple» in Lisp, and others [. . .] argue that design patterns amount to admissions of inexpressiveness in programming languages.

«Design Patterns as Higher-Order Datatype-Generic Program», *Gibbons (2006)*

Or, les patrons de conception sont souvent liés à des limites du langage utilisé

Deux citations

Norvig [wrote] that 16 of the 23 GOF patterns are «invisible or simple» in Lisp, and others [. . .] argue that design patterns amount to admissions of inexpressiveness in programming languages.

«Design Patterns as Higher-Order Datatype-Generic Program», *Gibbons (2006)*

[A]fter learning a bit about functional programming idioms, it seems many of the GOF patterns are OO hacks to emulate functional idioms.

`https://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures`

Code smells vs. Language smells

Code smell

«Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality.»

Source: `https://en.wikipedia.org/wiki/Code_smell`

Code smells vs. Language smells

Code smell

«Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality.»

Source: *https://en.wikipedia.org/wiki/Code_smell*

Language smell

A `LanguageSmell` *is a* `CodeSmell` *that occurs due to the inability of the language to express the concept cleanly. Sometimes called an «idiom» or a «pattern.»*

Source: *https://wiki.c2.com/?LanguageSmell*

Les patrons de conception «à la GoF» sont intimement liés aux langages et concepts OOP

Comment les patrons de conception peuvent-ils être représentés dans un langage fonctionnel ?

Comment les patrons de conception peuvent-ils être représentés dans un langage fonctionnel ?

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• Yes, functions
• Strategy pattern	• Oh my, functions again!
• Decorator pattern	• Functions
• Visitor pattern	• Functions

Le patron *Composite* ⋆

Le patron *Composite*

Le problème

The Composite design pattern solves problems like : How can a part-whole hierarchy be represented so that clients can treat individual objects and compositions of objects uniformly ?

`https://w3sdesign.com/?gr=b09&ugr=proble#gf`

On peut utiliser un type algébrique pour définir un type `Composite` et une expression `case` pour le traiter

Elm : L'exemple de l'arbre binaire

```
type Arbre t
   = Feuille t
   | Noeud t (Arbre t) (Arbre t)


foo : Arbre t -> r
foo arbre =
  case arbre of
    Feuille v -> ...

    Noeud v gauche droite -> ...
```

Le patron *Strategy* ⋆

Le patron *Strategy*

Le problème

The Strategy design pattern solves problems like : How can a class be configured with an algorithm at run-time instead of implementing an algorithm directly ?

`https://w3sdesign.com/?gr=b09&ugr=proble#gf`

Une fonction d'ordre supérieur peut être utilisée pour paramétriser l'algorithme à utiliser

Elm

```
type alias MethodeDeTri comparable =
    List comparable -> List comparable

trier : MethodeDeTri comparable -> List comparable
                                -> List comparable
trier methodeTri = methodeDeTri

triSelection : MethodeDeTri comparable
triSelection = ...

triRapide : MethodeDeTri comparable
triRapide = ...

r1 = l1 |> trier triSelection
r2 = l1 |> trier triRapide
```

Le patron *Template method*

Le patron *Template method*

Le problème

The Template Method design pattern solves problems like : How can the invariant parts of a behavior be implemented once so that subclasses can implement the variant parts ?

`https://w3sdesign.com/?gr=b09&ugr=proble#gf`

Exemple : Un algorithme «diviser-pour-régner» générique

Elm

```
type alias StrategieDPR probleme solution =
  { estSimple : probleme -> Bool
  , resoudreSimple : probleme -> solution
  , decomposer : probleme -> List probleme
  , combiner : probleme -> List solution -> solution
  }
```

Exemple : Un algorithme «diviser-pour-régner» générique

Elm

```
resoudreDPR : StrategieDPR probleme solution
              -> probleme
              -> solution
resoudreDPR str leProbleme =

 if str.estSimple leProbleme then
   str.resoudreSimple leProbleme

 else
   let
     sousProblemes =
       str.decomposer leProbleme
     solutionsSousProblemes =
       map (resoudreDPR str) sousProblemes
   in
   str.combiner leProbleme solutionsSousProblemes
```

Exemple : Un algorithme «diviser-pour-régner» générique et son instantiation pour `fibonacci`

Elm

```
fibonacci : number -> number
fibonacci =
   let
     strategieFibo =
       { estSimple = \n -> n ≤ 1
       , resoudreSimple = always 1
       , decomposer = \n -> [ n - 1, n - 2 ]
       , combiner = \_ -> reduce (+) 0
       }
   in
   resoudreDPR strategieFibo
```

Exemple : Un algorithme «diviser-pour-régner» générique et son instantiation pour la `somme` d'un arbre

Elm

```
somme : Arbre.Arbre number -> number
somme =
  let
    strategieSomme =
      { estSimple = Arbre.estFeuille
      , resoudreSimple = Arbre.valeur
      , decomposer = Arbre.noeuds
      , combiner = \a -> reduce (+) (Arbre.valeur a)
      }
   in
   resoudreDPR strategieSomme
```

Exemple : Un algorithme «diviser-pour-régner» générique et son instantiation pour le `triFusion`

Elm

```
triFusion : List comparable -> List comparable
triFusion =
  let
    strategieTF =
      { estSimple = \l -> length l ≤ 1
      , resoudreSimple = identity
      , decomposer = \decouperEnDeux
      , combiner = \_ -> fusionner
      }
  in
  resoudreDPR strategieTF
```

Aperçu

Quelques avantages des langages fonctionnels

- Expressivité et concision !

Quelques avantages des langages fonctionnels

- Expressivité et concision !
- Avec typage statique fort et inférence de types
  - Quand ça compile, ça marche !

    «a good static type system is like having compile-time unit tests»

  - Facilite le réusinage du code (*refactoring*)

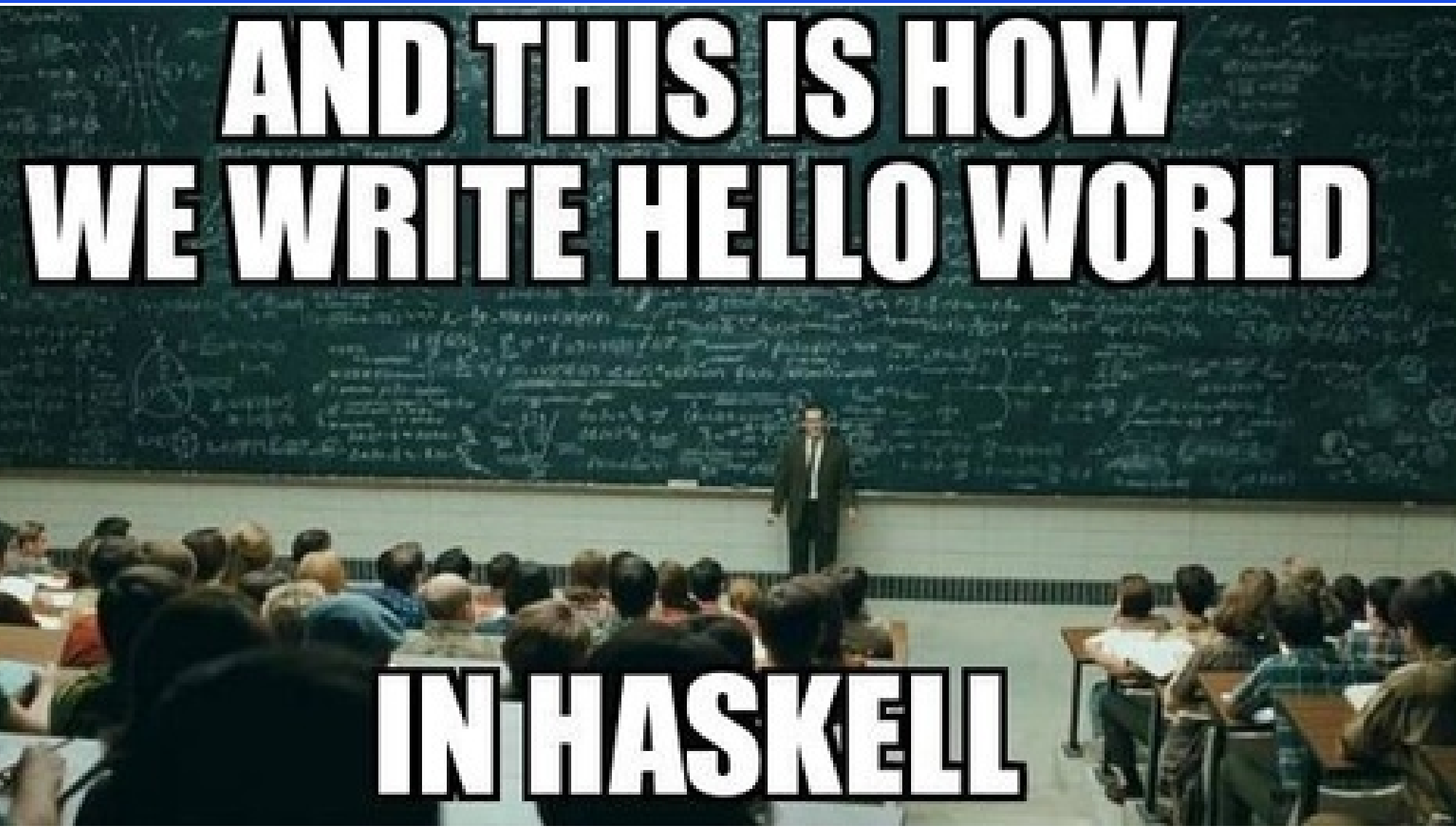
Quelques avantages des langages fonctionnels

- Expressivité et concision !
- Avec typage statique fort et inférence de types
  - Quand ça compile, ça marche !

    «a good static type system is like having compile-time unit tests»

  - Facilite le réusinage du code (*refactoring*)
- Facilite l'exploitation de la concurrence :
  L'immutabilité prévient les problèmes d'interférence (pas de variables partagées modifiées par plusieurs *threads*) !

Désavantages



`Hello World!` en Haskell

Haskell : `cat hello.hs`

```
module Main where

main = do
     print "Hello World!"
```

Hello World! en Haskell vs. en Java

Haskell : cat hello.hs

```haskell
module Main where

main = do
     print "Hello World!"
```

Java : cat Hello.java

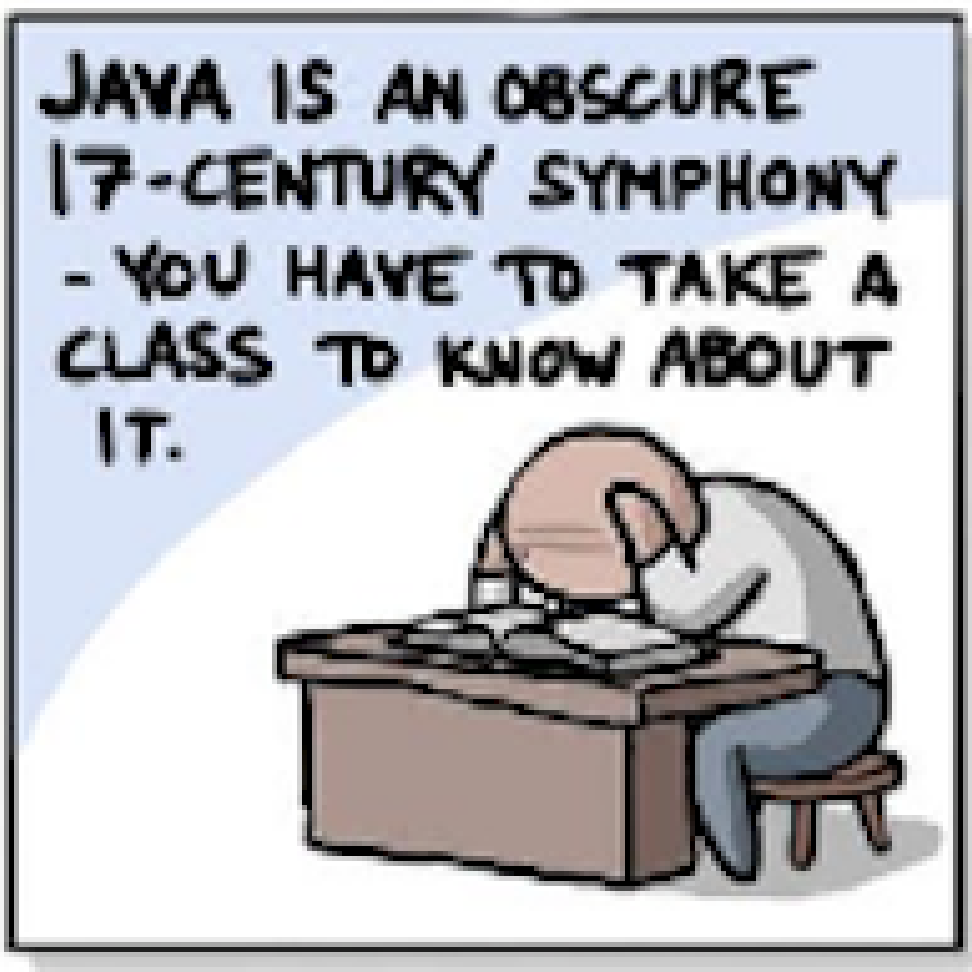
```java
public class Hello {
    public static void main(String[] args) {
        System.out.println( "Hello World!" );
    }
}
```

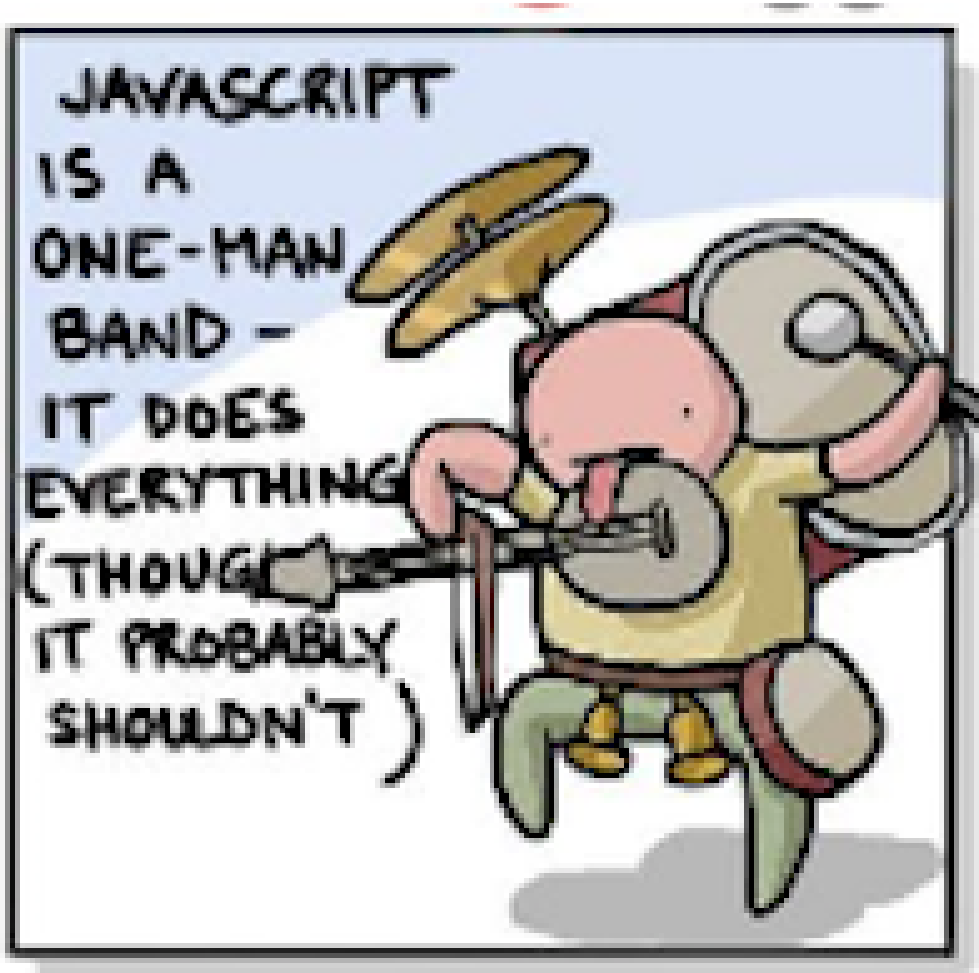
Désavantages

Programming explained with Music (Mart Virkus, 2019)

Désavantages

Programming explained with Music (Mart Virkus, 2019)





Pourquoi apprendre de nouveaux langages...

Pourquoi apprendre de nouveaux langages. . . fonctionnels, surtout si vous n'en connaissez pas ?

«A language that doesn't affect the way you think about programming is not worth knowing.»

A.J. Perlis

Pourquoi apprendre de nouveaux langages. . . fonctionnels, surtout si vous n'en connaissez pas ?

«A language that doesn't affect the way you think about programming is not worth knowing.»

A.J. Perlis

«If you can pick up a language without learning much, it means the language does not have much to teach !»

?

Deux langages intéressants que j'ai explorés cet hiver

Elixir : J. Valim (2011) = `https://elixir-lang.org/`

- \+ Conçu pour la concurrence et la robustesse — Erlang/OTP
- \+ Syntaxe flexible $\approx$ Ruby
- \+ Définition de fonctions multi-clauses via *pattern-matching*
- − Pas de types algébriques, pas de généricité — `tuples`, ..., `tuples`
- − Typage dynamique — sinon vérif. statique avec outil externe et annotations (plutôt lourdes)
- − Appel de fonction $\neq$ Appel d'expression lambda

Deux langages intéressants que j'ai explorés cet hiver

Elm : E. Czaplicki (2012) = `https://elm-lang.org/`

- \+ $\approx$ Haskell simplifié (e.g., pas de *type class*) et, surtout, **strict** (plutôt que paresseux)
- \+ Messages d'erreur du compilateur significatifs et compréhensibles
- \+ Super débogueur
- \+ Architecture *Model/View/Update* pour applications Web ($\Rightarrow$ JavaScript)
- − Langage (et son concepteur BDFL) très opiniâtre
- − Encore en évolution : 0.18 $>$ 0.19

Pour en savoir plus. . .


J. Fairbank.
Programming Elm : Build Safe, Sane, and Maintainable Front-End Applications.
Pragmatic Bookshelf, 2019.


L. Halvorsen.
Functional Web Development with Elixir, OTP, and Phoenix : Rethink the Modern Web App
Pragmatic Bookshelf, 2018.


B.J. MacLennan.
Values and objects in programming languages.
SIGPLAN Notices, 17(12) :70–79, 1982.


D. Thomas.
Programming Elixir Functional $|>$*Concurrent* $|>$*Pragmatic* $|>$*Fun*.
Pragmatic Bookshelf, 2014.


S. Waschlin
Domain Modeling Made Functional—Tackling Software Complexity with Domain-Driven Design and F#
Pragmatic Bookshelf, 2018.

Questions ?

Commentaires ?

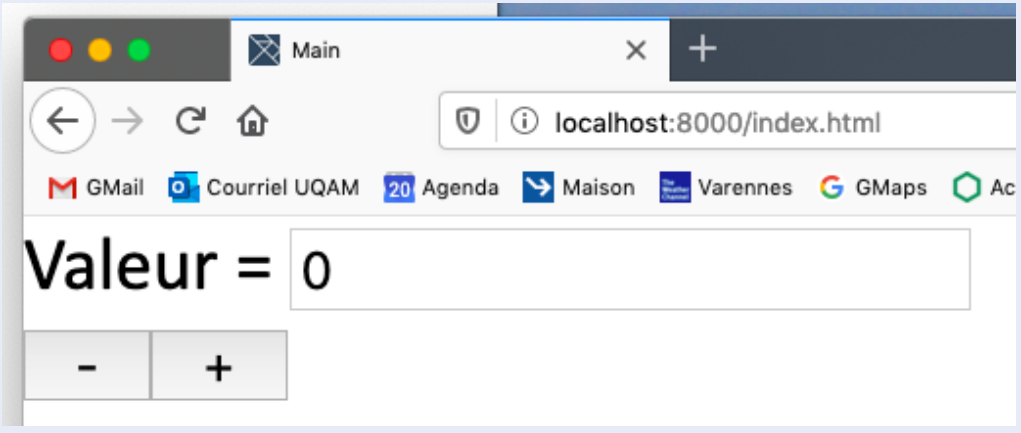
A. Quelques jalons dans l'évolution des langages fonctionnels

1958	Lisp	Fonct. d'ordre sup. (`maplist`★), λ-expr.★, récursion
1972	SASL	*Layout (offside) rule* ★
1973	ML	Polymorphisme paramétrique, inférence de types★, filtrage de motifs (*pattern-match.*) des args., currying
1976	SASL	Paresseux (U. de M. *circa* 1980)
1978	FP	Sans variable (*point free*))
1978	Id	Implicitement parallèle
1979	KRC	Gardes, ZF-expressions★(*list comprehensions*)
1980	Hope	Types algébriques de données★
1985	SISAL	*Streams*★
1985	Miranda	$\approx$ SASL + KRC + Types algébriques p
1990	Haskell	Miranda + ML + *type classes* + etc. + etc. (Designed by a commitee)
1993	pH	Haskell + Id (Implicitement parallèle)
2005	F#	ML + C# (Microsoft)
2007	Clojure	Lisp sur la JVM (Un peu moins de «(. . .)»)
2011	Elixir	Erlang + syntaxe style Ruby
2012	Elm	Haskell simplifié (sans *type classes* ou sucre syntaxique)

B. Un petit exemple d’application Web avec Elm

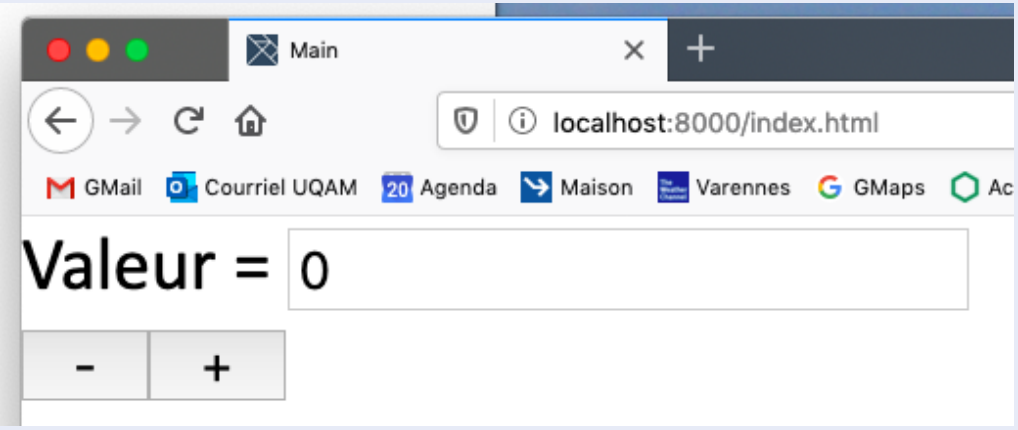
Une «application» Web pour un compteur entier avec trois opérations : `Increment`, `Decrement` et `Set`

État initial

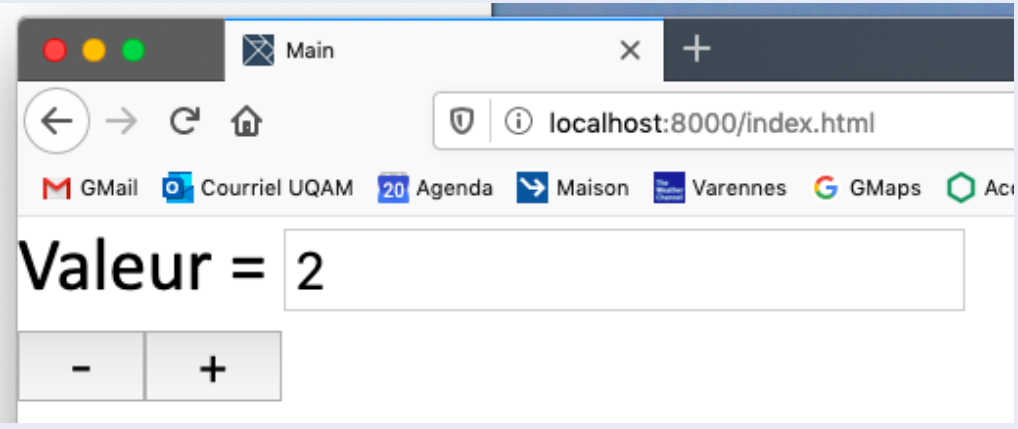


Une «application» Web pour un compteur entier avec trois opérations : `Increment`, `Decrement` et `Set`

État initial



Après avoir cliqué deux fois le bouton «+»



Elm : Les éléments importés et les types définis puis utilisés dans le programme

```
% cat src/Main.elm
module Main exposing (main)

import Html exposing (Attribute, Html)
import Html.Attributes
import Html.Events
import String


type alias Model =
    Int

type Msg
    = Increment
    | Decrement
    | Set String
```

Elm : Le programme principal

```
main =
    Html.beginnerProgram
        { model = 0          -- Etat initial
        , update = update
        , view = view
        }
```

Elm : La fonction pour la vue, qui utilise le module `Html`

```
view : Model -> Html Msg
view model =
  Html.div []
    [ Html.div []
      [ Html.text "Valeur = "
      , Html.input
        [ Html.Attributes.placeholder "0"
        , Html.Attributes.value (toString model)
        , Html.Events.onInput Set
        ] []
      ]
      , Html.button
        [ Html.Events.onClick Decrement ]
        [ Html.text "-" ]
      , Html.button
        [ Html.Events.onClick Increment ]
        [ Html.text "+" ]
    ]
```

Elm : La fonction pour la mise à jour de l'état de l'application, suite à la réception d'un des messages

```
update : Msg -> Model -> Model
-- Msg: Message recu de l'environnement
-- ->
-- Model: Etat courant
-- ->
-- Model: Nouvel etat
--
update msg model =
    case msg of -- Etat courant
        Increment ->
            model + 1  -- Nouvel etat

        Decrement ->
            model - 1

        Set str ->
            str
              |> String.toInt
              |> Result.withDefault model
```