

# Programmation fonctionnelle

**Samuele Girardo**

Université Paris-Est Marne-la-Vallée

LIGM, bureau 4B055

`samuele.girardo@u-pem.fr`

`http://igm.univ-mlv.fr/~girardo/`

LES ARBRES BINAIRES DE RECHERCHE  
EN  
PROGRAMMATION FONCTIONNELLE

# Arbres binaires

Un **arbre binaire** est

# Arbres binaires

Un **arbre binaire** est

**1** soit une feuille



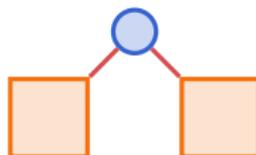
# Arbres binaires

Un **arbre binaire** est

1 soit une feuille



2 soit un nœud attaché à deux arbres binaires



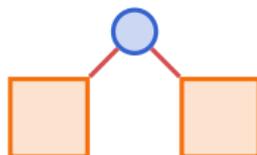
# Arbres binaires

Un **arbre binaire** est

1 soit une feuille



2 soit un nœud attaché à deux arbres binaires



C'est une définition **récursive** car la définition de la notion fait référence à la notion elle-même.

# Arbres binaires en Caml

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

# Arbres binaires en Caml

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

Définition du type arbre binaire en Caml :

```
type arbre_b =  
  Feuille  
  | Noeud of arbre_b * int * arbre_b;;
```

# Arbres binaires en Caml

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

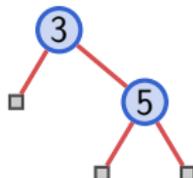
**Définition du type** arbre binaire en Caml :

```
type arbre_b =  
  Feuille  
  | Noeud of arbre_b * int * arbre_b;;
```

**Construction** d'un arbre binaire en Caml :

```
Noeud (Feuille, 3, Noeud (Feuille, 5, Feuille))
```

Cette expression désigne l'arbre binaire



# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

Cet arbre se définit de la manière suivante :

# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

Cet arbre se définit de la manière suivante :

1 si  $T = \square$ , alors



# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

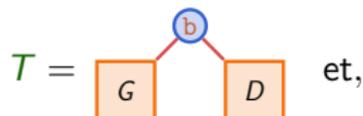
On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

Cet arbre se définit de la manière suivante :

1 si  $T = \square$ , alors



2 sinon,  $T$  est de la forme



# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

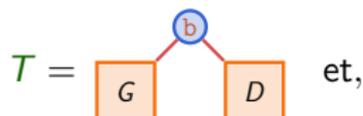
On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

Cet arbre se définit de la manière suivante :

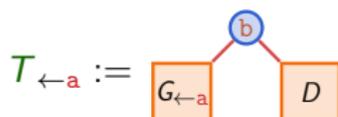
1 si  $T = \square$ , alors



2 sinon,  $T$  est de la forme



■ si  $a \leq b$ , alors



# Insertion dans un arbre binaire de recherche

Soit  $T$  arbre binaire de recherche.

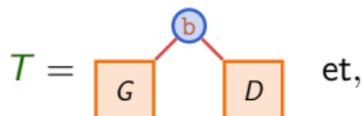
On note  $T_{\leftarrow a}$  l'arbre binaire obtenu après **insertion** de la lettre  $a$  dans  $T$ .

Cet arbre se définit de la manière suivante :

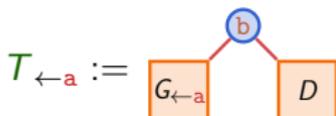
1 si  $T = \square$ , alors



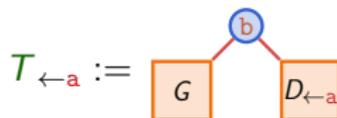
2 sinon,  $T$  est de la forme



■ si  $a \leq b$ , alors



■ sinon  $a > b$  et



# Insertion dans un arbre binaire de recherche en Caml

**Définition de la fonction** (récursive) d'insertion dans un arbre binaire de recherche en Caml :

```
let rec inserer t a =  
  match t with  
  |Feuille -> Noeud (Feuille, a, Feuille)  
  |Noeud (g, b, d) when a <= b -> Noeud (inserer g a, b, d)  
  |Noeud (g, b, d) -> Noeud (g, b, inserer d a);;
```

# Insertion dans un arbre binaire de recherche en Caml

**Définition de la fonction** (réursive) d'insertion dans un arbre binaire de recherche en Caml :

```
let rec inserer t a =  
  match t with  
  |Feuille -> Noeud (Feuille, a, Feuille)  
  |Noeud (g, b, d) when a <= b -> Noeud (inserer g a, b, d)  
  |Noeud (g, b, d) -> Noeud (g, b, inserer d a);;
```

**Construction** d'un arbre binaire de recherche par insertions successives :

```
let t1 = inserer Feuille 3 in  
let t2 = inserer t1 4 in  
let t3 = inserer t2 2 in  
inserer t3 1
```

# INSTRUCTIONS *vs* EXPRESSIONS

## Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

# Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le paradigme impératif, on propose (en Python),

```
def f(lst) :  
    res = []  
    for i in range(len(lst)) :  
        if i % 2 == 0 :  
            res.append(lst[i])  
    return res
```

# Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le paradigme impératif, on propose (en Python),

```
def f(lst) :  
    res = []  
    for i in range(len(lst)) :  
        if i % 2 == 0 :  
            res.append(lst[i])  
    return res
```

Le calcul de `f([0, 1, 2, 3, 4])` utilise de la mémoire :

- pour construire le résultat `res` ;
- pour maintenir la variable `i` ;
- éventuellement pour les fonctions appelées.

Les variables `res` et `i` sont lues et modifiées au cours du temps.

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =
  if (List.length lst) <= 1 then
    lst
  else
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'expression `f([0 ; 1 ; 2 ; 3 ; 4])`, on l'évalue :

`f([0; 1; 2; 3; 4])`

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'**expression** `f([0 ; 1 ; 2 ; 3 ; 4])`, on l'**évalue** :

`f([0 ; 1 ; 2 ; 3 ; 4])`  $\rightarrow$  `0 :: f([2 ; 3 ; 4])`

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'expression  $f([0; 1; 2; 3; 4])$ , on l'évalue :

$$f([0; 1; 2; 3; 4]) \rightarrow 0 :: f([2; 3; 4]) \rightarrow 0 :: 2 :: f([4])$$

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'expression  $f([0; 1; 2; 3; 4])$ , on l'évalue :

$$f([0; 1; 2; 3; 4]) \rightarrow 0 :: f([2; 3; 4]) \rightarrow 0 :: 2 :: f([4]) \rightarrow 0 :: 2 :: [4]$$

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =  
  if (List.length lst) <= 1 then  
    lst  
  else  
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'**expression**  $f([0; 1; 2; 3; 4])$ , on l'**évalue** :

$$\begin{aligned} f([0; 1; 2; 3; 4]) &\rightarrow 0 :: f([2; 3; 4]) \rightarrow 0 :: 2 :: f([4]) \rightarrow 0 :: 2 :: [4] \\ &\rightsquigarrow [0; 2; 4] \end{aligned}$$

# Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en Caml de débutant) :

```
let rec f lst =
  if (List.length lst) <= 1 then
    lst
  else
    (List.hd lst) :: (f (List.tl (List.tl lst)));;
```

Pour calculer l'**expression**  $f([0; 1; 2; 3; 4])$ , on l'**évalue** :

$$\begin{aligned} f([0; 1; 2; 3; 4]) &\rightarrow 0 :: f([2; 3; 4]) \rightarrow 0 :: 2 :: f([4]) \rightarrow 0 :: 2 :: [4] \\ &\rightsquigarrow [0; 2; 4] \end{aligned}$$

Dans ce cas, il n'y a

- ni utilisation externe de la mémoire;
- ni état d'avancement du calcul qui dépend du temps.

Le calcul se fait en **réécriv**ant l'expression tant que possible. La définition de **fonctions** permet d'expliquer comment réaliser les réécritures.

# PRINCIPES GÉNÉRAUX

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

- Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

# Quelques caractéristiques du Caml

- **Allocation** de mémoire **implicite**.

↪ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

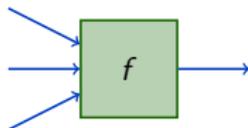
↪ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

- Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

↪ « *Le compilateur Caml n'accepte presque jamais mon code mais quand il l'accepte, ça marche; le compilateur \* accepte presque toujours mon code mais ça ne marche presque jamais.* »

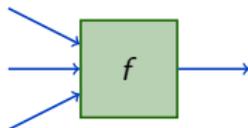
# Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



# Principes de la programmation fonctionnelle

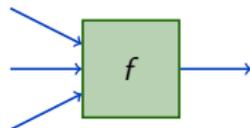
La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



On évite les **effets de bord** (et donc, on ne fait pas d'affectation).

# Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :

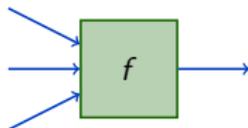


On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

On **évite les séquences d'instructions impératives** (et donc, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

# Principes de la programmation fonctionnelle

La **programmation fonctionnelle** est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

On **évite les séquences d'instructions impératives** (et donc, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

On utilise en revanche l'**application de fonctions** et la **récurtivité**.

Ce cours demande les pré-réquis suivants :

- des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets) ;

Ce cours demande les pré-réquis suivants :

- des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets) ;
- des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord) ;

Ce cours demande les pré-réquis suivants :

- des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets) ;
- des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord) ;
- des connaissances solides du **langage C** ou du **langage Python** ;

Ce cours demande les pré-réquis suivants :

- des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets) ;
- des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets de bord) ;
- des connaissances solides du **langage C** ou du **langage Python** ;
- des connaissances de base en **algorithmique** (récursivité, manipulation de listes, d'arbres).

# Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

- 1 d'avoir une 1<sup>re</sup> approche du **paradigme de programmation fonctionnelle** ;
- 2 d'apprendre les bases de **programmation en Caml** en mode fonctionnel.

# Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

- 1 d'avoir une 1<sup>re</sup> approche du **paradigme de programmation fonctionnelle** ;
- 2 d'apprendre les bases de **programmation en Caml** en mode fonctionnel.

Celui-ci est organisé en trois axes.

**1 Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

# Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

- 1 d'avoir une 1<sup>re</sup> approche du **paradigme de programmation fonctionnelle** ;
- 2 d'apprendre les bases de **programmation en Caml** en mode fonctionnel.

Celui-ci est organisé en trois axes.

**1 Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

**2 Axe 2 : concepts premiers.**

Programmation en Caml, liaisons, fonctions, entrées/sorties, compilation, types.

# Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

- 1 d'avoir une 1<sup>re</sup> approche du **paradigme de programmation fonctionnelle** ;
- 2 d'apprendre les bases de **programmation en Caml** en mode fonctionnel.

Celui-ci est organisé en trois axes.

**1 Axe 1 : bases théoriques.**

Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

**2 Axe 2 : concepts premiers.**

Programmation en Caml, liaisons, fonctions, entrées/sorties, compilation, types.

**3 Axe 3 : concepts avancés.**

Récursivité terminale, filtrage, fonctions d'ordre supérieur, polymorphisme, stratégies d'évaluation, opérations sur les listes, non mutabilité,  $\lambda$ -calcul.

## Axe 1.

- 1 Théorie
- 2 Caractéristiques

## Axe 2.

- 3 Programmation
- 4 Pratique
- 5 Types

## Axe 3.

- 6 Notions
- 7 Listes
- 8  $\lambda$ -calcul

## Bibliographie non exhaustive :

- X. Leroy, P. Weis, *Le langage Caml*, Dunod, 2<sup>e</sup> édition, 1999.  
Lien : <http://caml.inria.fr/distrib/books/llc.pdf>
- E. Chailloux, P. Manoury, B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly, 2000.  
Lien : <http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML/>
- G. Dowek, J.-J. Lévy, *Introduction à la théorie des langages de programmation*, École Polytechnique, 2006.
- C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1999.
- S. L. Peyton Jones, D. Lester, *Implementing Functional Languages*, Prentice Hall, 1992.

# Axe 1 : bases théoriques

1 Théorie

2 Caractéristiques

- 1 Théorie
  - Machines de Turing
  - Décidabilité et indécidabilité
  - $\lambda$ -calcul

- 1 Théorie
  - Machines de Turing
  - Décidabilité et indécidabilité
  - $\lambda$ -calcul

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.
- 1936 : Church introduit le  **$\lambda$ -calcul**.

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.
- 1936 : Church introduit le  **$\lambda$ -calcul**.
- 1936 : Turing invente la **machine de Turing**.

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.
- 1936 : Church introduit le  **$\lambda$ -calcul**.
- 1936 : Turing invente la **machine de Turing**.
- Années 1940 : programmation en assembleur et en langage machine.

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.
- 1936 : Church introduit le  **$\lambda$ -calcul**.
- 1936 : Turing invente la **machine de Turing**.
- Années 1940 : programmation en assembleur et en langage machine.
- Années 1950-1960 : 1<sup>ers</sup> vrais **langages de programmation** : FORTRAN, COBOL et LISP.

# Brève chronologie de la programmation

- 1801 : 1<sup>re</sup> machine programmable : le **métier à tisser** de Jackard.
- 1936 : Church introduit le  **$\lambda$ -calcul**.
- 1936 : Turing invente la **machine de Turing**.
- Années 1940 : programmation en assembleur et en langage machine.
- Années 1950-1960 : 1<sup>ers</sup> vrais **langages de programmation** : FORTRAN, COBOL et LISP.
- Années 1960-1970 : **paradigmes de programmation** : impératif, fonctionnel, orienté objet, logique.

# Machines de Turing

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

# Machines de Turing

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

# Machines de Turing

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet  $(E, i, t, \Delta)$  où

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet  $(E, i, t, \Delta)$  où

- 1  $E$  est un ensemble fini d'états ;

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet  $(E, i, t, \Delta)$  où

- 1  $E$  est un ensemble fini d'états ;
- 2  $i \in E$  est l'état initial ;

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet  $(E, i, t, \Delta)$  où

- 1  $E$  est un ensemble fini d'états ;
- 2  $i \in E$  est l'état initial ;
- 3  $t \in E$  est l'état terminal ;

**Machine de Turing** : machine théorique qui fournit une abstraction des appareils de calcul.

**Thèse de Church-Turing** :

« tout ce qui est effectivement *calculable*  
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet  $(E, i, t, \Delta)$  où

- 1  $E$  est un ensemble fini d'états ;
- 2  $i \in E$  est l'état initial ;
- 3  $t \in E$  est l'état terminal ;
- 4  $\Delta : E \times \{\cdot, 0, 1\} \rightarrow E \times \{\cdot, 0, 1\} \times \{G, D\}$  est une fonction de transition.

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

L'**exécution** de  $M$  sur  $u$  consiste à :

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

L'**exécution** de  $M$  sur  $u$  consiste à :

- 1 placer  $u$  le plus à gauche dans un tableau infini à droite, le **ruban** :

$u_1$	$u_2$	$\dots$	$u_{ u }$	.	.	$\dots$
-------	-------	---------	-----------	---	---	---------

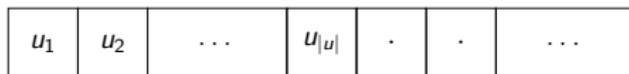
Les cases à droite de  $u$  sont remplies jusqu'à l'infini de ..

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

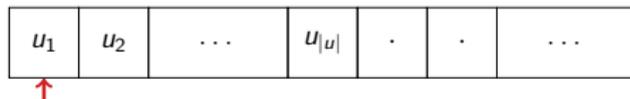
L'**exécution** de  $M$  sur  $u$  consiste à :

- 1 placer  $u$  le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de  $u$  sont remplies jusqu'à l'infini de .

- 2 Placer la **tête de lecture/écriture** sur la 1<sup>re</sup> case du ruban :



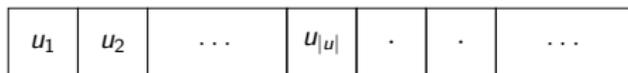
On appelle  $a$  la lettre dans  $\{., 0, 1\}$  indiquée par la tête de lecture/écriture à un instant donné.

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

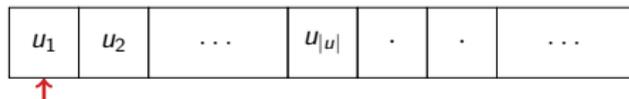
L'**exécution** de  $M$  sur  $u$  consiste à :

- 1 placer  $u$  le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de  $u$  sont remplies jusqu'à l'infini de .

- 2 Placer la **tête de lecture/écriture** sur la 1<sup>re</sup> case du ruban :



On appelle  $a$  la lettre dans  $\{\cdot, 0, 1\}$  indiquée par la tête de lecture/écriture à un instant donné.

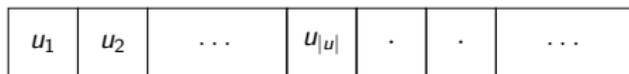
- 3 Affecter au **registre d'état**  $e$  la valeur  $i$  (état initial).

# Fonctionnement des machines de Turing

Soit  $M := (E, i, t, \Delta)$  une machine de Turing et  $u \in \{0, 1\}^*$  un mot.

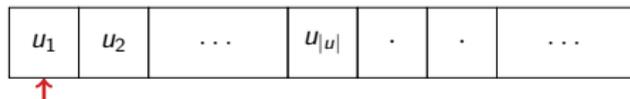
L'**exécution** de  $M$  sur  $u$  consiste à :

- 1 placer  $u$  le plus à gauche dans un tableau infini à droite, le **ruban** :



Les cases à droite de  $u$  sont remplies jusqu'à l'infini de .

- 2 Placer la **tête de lecture/écriture** sur la 1<sup>re</sup> case du ruban :



On appelle  $a$  la lettre dans  $\{., 0, 1\}$  indiquée par la tête de lecture/écriture à un instant donné.

- 3 Affecter au **registre d'état**  $e$  la valeur  $i$  (état initial).
- 4 Réaliser les actions dictées par  $\Delta$ , le **programme**.

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

(1) calculer  $(e', a', s') := \Delta(e, a)$ .

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

- (1) calculer  $(e', a', s') := \Delta(e, a)$ .
- (2) Écrire  $a'$  dans la case du ruban indiquée par la tête de lecture/écriture.

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

- (1) calculer  $(e', a', s') := \Delta(e, a)$ .
- (2) Écrire  $a'$  dans la case du ruban indiquée par la tête de lecture/écriture.
- (3) Affecter au registre d'état  $e$  l'état  $e'$ .

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

- (1) calculer  $(e', a', s') := \Delta(e, a)$ .
- (2) Écrire  $a'$  dans la case du ruban indiquée par la tête de lecture/écriture.
- (3) Affecter au registre d'état  $e$  l'état  $e'$ .
- (4) Si  $s' = D$ , déplacer la tête de lecture/écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

- (1) calculer  $(e', a', s') := \Delta(e, a)$ .
- (2) Écrire  $a'$  dans la case du ruban indiquée par la tête de lecture/écriture.
- (3) Affecter au registre d'état  $e$  l'état  $e'$ .
- (4) Si  $s' = D$ , déplacer la tête de lecture/écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.
- (5) Si  $e = t$ , alors l'exécution est terminée ; sinon, revenir en (1).

# Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de  $M$ , on procède séquentiellement comme suit :

- (1) calculer  $(e', a', s') := \Delta(e, a)$ .
- (2) Écrire  $a'$  dans la case du ruban indiquée par la tête de lecture/écriture.
- (3) Affecter au registre d'état  $e$  l'état  $e'$ .
- (4) Si  $s' = D$ , déplacer la tête de lecture/écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.
- (5) Si  $e = t$ , alors l'exécution est terminée ; sinon, revenir en (1).

**Résultat**  $M(u)$  de l'exécution de  $M$  sur  $u$  : plus court préfixe du ruban qui contient tous ses 0 et ses 1.

## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, 0)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

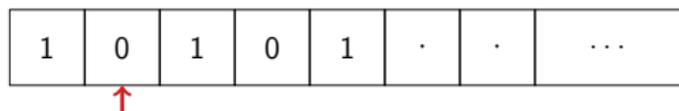
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, 0)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, 1)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, 0)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

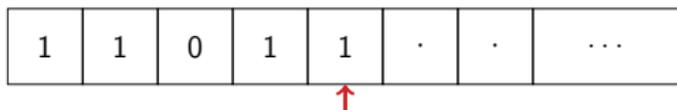
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, 1)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

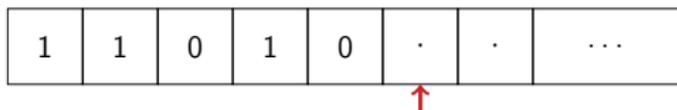
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_1, \cdot)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

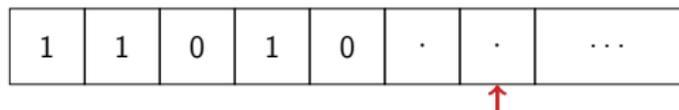
$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_2, \cdot)$$



## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 1, D),$$

$$(e_1, 1) \mapsto (e_1, 0, D),$$

$$(e_1, \cdot) \mapsto (e_2, \cdot, D).$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_2, \cdot)$$

1	1	0	1	0	.	.	...
---	---	---	---	---	---	---	-----

L'exécution de  $M$  sur  $u$  fournit ainsi le résultat 11010.

## Exemple : complémentaire d'un mot

Soit  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  la machine de Turing dont le programme  $\Delta$  est défini par

$$\begin{aligned}(e_1, 0) &\mapsto (e_1, 1, D), \\(e_1, 1) &\mapsto (e_1, 0, D), \\(e_1, \cdot) &\mapsto (e_2, \cdot, D).\end{aligned}$$

Calcul de  $M(00101)$  :

$$(e, a) = (e_2, \cdot)$$

1	1	0	1	0	.	.	...
---	---	---	---	---	---	---	-----

L'exécution de  $M$  sur  $u$  fournit ainsi le résultat 11010.

Ici, ce programme  $\Delta$  permet de calculer le complémentaire de tout mot  $u \in \{0, 1\}^*$  en entrée.

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

# Exécutions qui ne se terminent pas

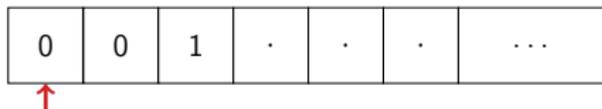
Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_1, 0)$$



# Exécutions qui ne se terminent pas

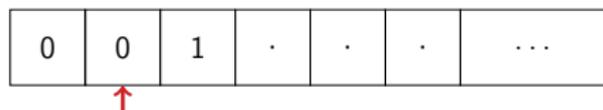
Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_1, 0)$$



# Exécutions qui ne se terminent pas

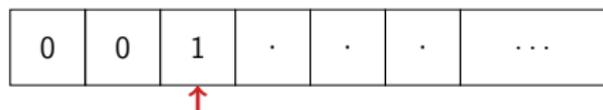
Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_1, 1)$$



# Exécutions qui ne se terminent pas

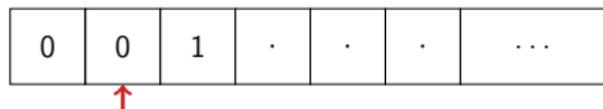
Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, 0)$$



# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, 0)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----

↑

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, 0)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----

↑

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, 0)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----



# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, \cdot)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----



# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, \cdot)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----

↑

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, \cdot)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----

↑

# Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées  $u$ , l'exécution de  $M$  sur  $u$  **ne se termine pas**.

Soit par exemple la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

Calcul de  $M(001)$  :

$$(e, a) = (e_2, \cdot)$$

0	0	1	.	.	.	...
---	---	---	---	---	---	-----

On arrive à  $e_2$  qui est l'état terminal, l'exécution est terminée.

Calcul de  $M(000)$  :

$$(e, a) = (e_1, \cdot)$$

0	0	0	.	.	.	...
---	---	---	---	---	---	-----

La tête de lecture/écriture part vers infini, l'exécution ne se termine pas.

# Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing**  $M$  un **entier naturel**  $\text{code}(M)$  de la manière suivante :

# Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing**  $M$  un **entier naturel**  $\text{code}(M)$  de la manière suivante :

- 1 on fixe des conventions pour écrire les définitions des machines de Turing avec des caractères ASCII ;

# Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing**  $M$  un **entier naturel**  $\text{code}(M)$  de la manière suivante :

- 1 on fixe des conventions pour écrire les définitions des machines de Turing avec des caractères ASCII ;
- 2 on considère la suite de caractères  $m$  ainsi obtenue qui code  $M$  ;

# Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing**  $M$  un **entier naturel**  $\text{code}(M)$  de la manière suivante :

- 1 on fixe des conventions pour écrire les définitions des machines de Turing avec des caractères ASCII ;
- 2 on considère la suite de caractères  $m$  ainsi obtenue qui code  $M$  ;
- 3 on considère la suite de bits  $u$  obtenue en remplaçant chaque caractère de  $m$  par sa représentation binaire ;

# Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing**  $M$  un **entier naturel**  $\text{code}(M)$  de la manière suivante :

- 1 on fixe des conventions pour écrire les définitions des machines de Turing avec des caractères ASCII ;
- 2 on considère la suite de caractères  $m$  ainsi obtenue qui code  $M$  ;
- 3 on considère la suite de bits  $u$  obtenue en remplaçant chaque caractère de  $m$  par sa représentation binaire ;
- 4 on obtient finalement l'entier naturel  $\text{code}(M)$  en considérant l'entier dont  $u$  est la représentation binaire.

# Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

# Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

$M$  se code en caractères ASCII en le texte  $m$  suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

# Coder une machine de Turing par un entier naturel

Par exemple, considérons la machine de Turing  $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$  dont le programme  $\Delta$  est défini par

$$(e_1, 0) \mapsto (e_1, 0, D), \quad (e_1, 1) \mapsto (e_2, 1, G), \quad (e_1, \cdot) \mapsto (e_1, \cdot, D).$$

$M$  se code en caractères ASCII en le texte  $m$  suivant :

```
etats : e1, e2
initial : e1
terminal : e2
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

On en déduit la représentation binaire

$$u = 0110010101110100 \dots 0010100100001101$$

et de celle-ci, l'entier naturel  $\text{code}(M)$ .

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose  $M \leq M'$  si  $\text{code}(M) \leq \text{code}(M')$ .

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose  $M \leq M'$  si  $\text{code}(M) \leq \text{code}(M')$ .

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose  $M \leq M'$  si  $\text{code}(M) \leq \text{code}(M')$ .

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang**  $\text{rang}(M)$  d'une machine de Turing  $M$  est la position de  $M$  dans ce segment.

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose  $M \leq M'$  si  $\text{code}(M) \leq \text{code}(M')$ .

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang**  $\text{rang}(M)$  d'une machine de Turing  $M$  est la position de  $M$  dans ce segment.

L'application `rang` fournit une bijection entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

# Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application `code` est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose  $M \leq M'$  si  $\text{code}(M) \leq \text{code}(M')$ .

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang**  $\text{rang}(M)$  d'une machine de Turing  $M$  est la position de  $M$  dans ce segment.

L'application `rang` fournit une bijection entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

**Conclusion** : un programme (une machine de Turing) est un entier et réciproquement.

- 1 Théorie
  - Machines de Turing
  - Décidabilité et indécidabilité
  - $\lambda$ -calcul

**Problème de décision** : question  $P$  qui prend un mot de  $\{0, 1\}^*$  en entrée et qui répond oui ou non.

# Problèmes de décision

**Problème de décision** : question  $P$  qui prend un mot de  $\{0,1\}^*$  en entrée et qui répond oui ou non.

Par exemple :

- $P$  : le mot est-il un palindrome ?  $P(010010) = \text{oui}$ ,  $P(011) = \text{non}$  ;

# Problèmes de décision

**Problème de décision** : question  $P$  qui prend un mot de  $\{0, 1\}^*$  en entrée et qui répond oui ou non.

Par exemple :

- $P$  : le mot est-il un palindrome ?  $P(010010) = \text{oui}$ ,  $P(011) = \text{non}$  ;
- $P$  : la longueur du mot est-elle paire ?  $P(\epsilon) = \text{oui}$ ,  $P(1) = \text{non}$  ;

# Problèmes de décision

**Problème de décision** : question  $P$  qui prend un mot de  $\{0, 1\}^*$  en entrée et qui répond oui ou non.

Par exemple :

- $P$  : le mot est-il un palindrome ?  $P(010010) = \text{oui}$ ,  $P(011) = \text{non}$  ;
- $P$  : la longueur du mot est-elle paire ?  $P(\epsilon) = \text{oui}$ ,  $P(1) = \text{non}$  ;
- $P$  : l'entier en base deux codé par le mot est-il premier ?  
 $P(111) = \text{oui}$ ,  $P(100) = \text{non}$  ;

# Problèmes de décision

**Problème de décision** : question  $P$  qui prend un mot de  $\{0, 1\}^*$  en entrée et qui répond oui ou non.

Par exemple :

- $P$  : le mot est-il un palindrome ?  $P(010010) = \text{oui}$ ,  $P(011) = \text{non}$  ;
- $P$  : la longueur du mot est-elle paire ?  $P(\epsilon) = \text{oui}$ ,  $P(1) = \text{non}$  ;
- $P$  : l'entier en base deux codé par le mot est-il premier ?  
 $P(111) = \text{oui}$ ,  $P(100) = \text{non}$  ;
- $P$  : le mot est-il le codage binaire d'un programme  $C$  accepté à la compilation par `gcc` avec l'option `-ansi` ?

**Problème de décision décidable** : problème de décision  $P$  tel qu'il existe une machine de Turing  $M_P$  telle que pour toute entrée  $u \in \{0, 1\}^*$ , l'exécution de  $M_P$  sur  $u$  se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui,} \\ 0 & \text{sinon.} \end{cases}$$

**Problème de décision décidable** : problème de décision  $P$  tel qu'il existe une machine de Turing  $M_P$  telle que pour toute entrée  $u \in \{0, 1\}^*$ , l'exécution de  $M_P$  sur  $u$  se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui,} \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing,  $P$  est dit **indécidable**.

# Décidabilité et indécidabilité

**Problème de décision décidable** : problème de décision  $P$  tel qu'il existe une machine de Turing  $M_P$  telle que pour toute entrée  $u \in \{0, 1\}^*$ , l'exécution de  $M_P$  sur  $u$  se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui,} \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing,  $P$  est dit **indécidable**.

Intuitivement, un problème de décision  $P$  est décidable s'il est possible d'écrire, dans un langage suffisamment complet, une fonction  $f$  paramétrée par un objet  $u$  renvoyant `true` si  $P(u) = \text{oui}$  et `false` sinon.

# Le problème de l'arrêt

**Problème de l'arrêt** : problème de décision Arr prenant en entrée le codage binaire  $u$  d'un programme  $f$  et renvoyant oui si l'exécution du programme  $f$  se termine et non sinon.

# Le problème de l'arrêt

**Problème de l'arrêt** : problème de décision Arr prenant en entrée le codage binaire  $u$  d'un programme  $f$  et renvoyant oui si l'exécution du programme  $f$  se termine et non sinon.

Le problème de l'arrêt est **indécidable**.

# Le problème de l'arrêt

**Problème de l'arrêt** : problème de décision Arr prenant en entrée le codage binaire  $u$  d'un programme  $f$  et renvoyant oui si l'exécution du programme  $f$  se termine et non sinon.

Le problème de l'arrêt est **indécidable**.

Intuitivement, cela dit qu'il est impossible de concevoir un programme qui accepte en entrée un autre programme  $f$  et qui teste si l'exécution de  $f$  se termine.

# Indécidabilité du problème de l'arrêt

On montre que  $Arr$  est indécidable par l'absurde en supposant que  $Arr$  est décidable.

Il existe donc une machine de Turing  $M_{Arr}$ .

# Indécidabilité du problème de l'arrêt

On montre que  $\text{Arr}$  est indécidable par l'absurde en supposant que  $\text{Arr}$  est décidable.

Il existe donc une machine de Turing  $M_{\text{Arr}}$ .

Soit l'entier positif **absurde** défini par les instructions :

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

# Indécidabilité du problème de l'arrêt

On montre que  $\text{Arr}$  est indécidable par l'absurde en supposant que  $\text{Arr}$  est décidable.

Il existe donc une machine de Turing  $M_{\text{Arr}}$ .

Soit l'entier positif **absurde** défini par les instructions :

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

Intuitivement, **absurde** est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que **absurde**.

# Indécidabilité du problème de l'arrêt

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

# Indécidabilité du problème de l'arrêt

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

On vérifie facilement que l'exécution de **absurde** se termine.

# Indécidabilité du problème de l'arrêt

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

On vérifie facilement que l'exécution de **absurde** se termine.

Ainsi, la valeur de retour d'**absurde** figure dans  $E$  (étape (a)).

# Indécidabilité du problème de l'arrêt

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

On vérifie facilement que l'exécution de **absurde** se termine.

Ainsi, la valeur de retour d'**absurde** figure dans  $E$  (étape (a)).

Par ailleurs, la valeur renvoyée par **absurde** ne figure pas dans  $E$  (étape (3)).

# Indécidabilité du problème de l'arrêt

- (1) soit  $E$  l'ensemble vide ;
- (2) pour toute suite d'instructions  $f$  qui s'exprime avec moins ou autant de caractères que ces instructions définissant **absurde**,
  - (a) si  $M_{\text{Arr}}(u) = 1$ , où  $u$  est le codage binaire de  $f$ , ajouter à  $E$  la valeur calculée par  $f$ , exprimée par un entier.
- (3) Renvoyer  $\min \mathbb{N} \setminus E$ .

On vérifie facilement que l'exécution de **absurde** se termine.

Ainsi, la valeur de retour d'**absurde** figure dans  $E$  (étape (a)).

Par ailleurs, la valeur renvoyée par **absurde** ne figure pas dans  $E$  (étape (3)).

Ceci est absurde :  $M_{\text{Arr}}$  n'existe pas et Arr est donc indécidable.

- 1 Théorie
  - Machines de Turing
  - Décidabilité et indécidabilité
  - $\lambda$ -calcul

# Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

# Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Par exemple,

- $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$  ;
- $f : n \mapsto 1$  si  $n$  est premier, 0 sinon ;
- $f : n \mapsto 1$  si  $n \leq 1$ ,  $n \times f(n - 1)$  sinon ;

sont des fonctions récursives.

# Fonctions récursives

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Par exemple,

- $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$  ;
- $f : n \mapsto 1$  si  $n$  est premier, 0 sinon ;
- $f : n \mapsto 1$  si  $n \leq 1$ ,  $n \times f(n-1)$  sinon ;

sont des fonctions récursives.

En revanche, la fonction  $f : \mathbb{N} \rightarrow \mathbb{N}$  définie par

$$f(n) := \begin{cases} 1 & \text{si Arr}(g) = \text{oui où } g \text{ est le prog. tq. rang}(g) = n, \\ 0 & \text{sinon,} \end{cases}$$

n'est pas une fonction récursive (si elle était calculable, le problème de l'arrêt serait décidable).

# $\lambda$ -calcul

Le  $\lambda$ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

# $\lambda$ -calcul

Le  $\lambda$ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En  $\lambda$ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Le  $\lambda$ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En  $\lambda$ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du  $\lambda$ -calcul est

- 1 soit une **variable**, notée  $x, y, z, \dots$  ;
- 2 soit l'**application** d'une expression  $f$  à une expression  $g$ , notée  $fg$  ;
- 3 soit l'**abstraction** d'une expression  $f$ , notée  $\lambda x.f$  où  $x$  est une variable.

Le  $\lambda$ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En  $\lambda$ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du  $\lambda$ -calcul est

- 1 soit une **variable**, notée  $x, y, z, \dots$  ;
- 2 soit l'**application** d'une expression  $f$  à une expression  $g$ , notée  $fg$  ;
- 3 soit l'**abstraction** d'une expression  $f$ , notée  $\lambda x.f$  où  $x$  est une variable.

Par exemple,  $(\lambda z.z)((\lambda x.x)(\lambda y.((\lambda x.x)y)))$  est une expression.

Le  $\lambda$ -calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En  $\lambda$ -calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du  $\lambda$ -calcul est

- 1 soit une **variable**, notée  $x, y, z, \dots$ ;
- 2 soit l'**application** d'une expression  $f$  à une expression  $g$ , notée  $fg$ ;
- 3 soit l'**abstraction** d'une expression  $f$ , notée  $\lambda x.f$  où  $x$  est une variable.

Par exemple,  $(\lambda z.z)((\lambda x.x)(\lambda y.((\lambda x.x)y)))$  est une expression.

La  $\beta$ -**substitution** est le mécanisme qui permet de simplifier (calculer) une expression. Il consiste, étant donnée une expression de la forme  $(\lambda x.f)g$  à la simplifier en substituant  $g$  aux occurrences libres de  $x$  dans  $f$ .

- 2 Caractéristiques
  - Impératif vs fonctionnel
  - Caractéristiques des langages

- 2 Caractéristiques
  - Impératif vs fonctionnel
  - Caractéristiques des langages

# Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

# Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

# Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

- 1 le **paradigme impératif** ;
- 2 le **paradigme fonctionnel**.

# Paradigmes de programmation

Un **paradigme** est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un **paradigme de programmation** conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

- 1 le **paradigme impératif** ;
- 2 le **paradigme fonctionnel**.

Le 1<sup>er</sup> se base sur la machine de Turing, le 2<sup>e</sup> sur le  $\lambda$ -calcul.

# Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

# Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

- 1 les instructions d'affectation ;
- 2 les instructions de branchement ;
- 3 les instruction de boucle ;
- 4 les structures de données mutables.

# Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

- 1 les instructions d'affectation ;
- 2 les instructions de branchement ;
- 3 les instruction de boucle ;
- 4 les structures de données mutables.

Des instructions peuvent **modifier l'état de la machine** en altérant sa mémoire.

# Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

# Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Les éléments suivants sont constitutifs de ce paradigme :

- 1 la liaison d'un nom à une valeur ;
- 2 la récursivité ;
- 3 les instructions de branchement ;
- 4 les structures de données non mutables.

# Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Les éléments suivants sont constitutifs de ce paradigme :

- 1 la liaison d'un nom à une valeur ;
- 2 la récursivité ;
- 3 les instructions de branchement ;
- 4 les structures de données non mutables.

Il n'y a pas de notion d'état de la machine car celui-ci ne peut pas être modifié.

# Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

# Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}
```

```
int g(int n) {  
    return n;  
}
```

```
...  
g(f(1));
```

# Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}
```

```
int g(int n) {  
    return n;  
}
```

```
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche "a" mais pas `g(1)`.

# Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}
```

```
int g(int n) {  
    return n;  
}  
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche "a" mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

# Transparence référentielle

Principe de **transparence référentielle** : dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

Considérons le code C suivant :

```
int f(int n) {  
    printf("a");  
    return n;  
}
```

```
int g(int n) {  
    return n;  
}  
...  
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche "a" mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

**Règle** : en programmation fonctionnelle, le principe de transparence référentielle s'applique.

# Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

# Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

- 1 d'utiliser des variables (et donc aussi des affectations) ;
- 2 d'utiliser des instructions de boucle ;
- 3 de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

# Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

- 1 d'utiliser des variables (et donc aussi des affectations) ;
- 2 d'utiliser des instructions de boucle ;
- 3 de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

- 1 les **fonctions récursives** ;
- 2 les **fonctions locales**.

# Ce que l'on fera

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

- 1 d'utiliser des variables (et donc aussi des affectations) ;
- 2 d'utiliser des instructions de boucle ;
- 3 de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

- 1 les **fonctions récursives** ;
- 2 les **fonctions locales**.

Une variable peut-être vue comme une **fonction d'arité zéro** (c.-à-d. une fonction qui ne prend pas d'entrée).

- 2 **Caractéristiques**
  - Impératif vs fonctionnel
  - **Caractéristiques des langages**

# Typage dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

# Typage dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Il y a deux stratégies pour **vérifier les types** (typage) :

- 1 **typage dynamique**, où les types sont vérifiés lors de l'**exécution** du programme ;

# Typage dynamique vs statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Il y a deux stratégies pour **vérifier les types** (typage) :

- 1 **typage dynamique**, où les types sont vérifiés lors de l'**exécution** du programme ;
- 2 **typage statique**, où les types sont vérifiés lors de la **compilation** du programme.

# Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur 'n'.
    print n / 2    # Calcul arithm. sur 'n' et affichage.
else :
    print n[1]     # Affichage de la 1re case de 'n'.
```

# Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur 'n'.
    print n / 2    # Calcul arithm. sur 'n' et affichage.
else :
    print n[1]     # Affichage de la 1re case de 'n'.
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;

# Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur 'n'.
    print n / 2    # Calcul arithm. sur 'n' et affichage.
else :
    print n[1]     # Affichage de la 1re case de 'n'.
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;
- si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée ; sinon, c'est l'expression `n[1]` qui est évaluée.

# Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur 'n'.
    print n / 2    # Calcul arithm. sur 'n' et affichage.
else :
    print n[1]     # Affichage de la 1re case de 'n'.
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;
- si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée ; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

# Typage dynamique

Considérons le programme Python

```
n = int(input())    # Lect. de texte et conversion vers entier.
if n % 2 == 0 :    # Test de parité sur 'n'.
    print n / 2    # Calcul arithm. sur 'n' et affichage.
else :
    print n[1]     # Affichage de la 1re case de 'n'.
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;
- si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée ; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

Cette information n'est donnée que lors de l'exécution :

```
Traceback (most recent call last):
```

```
  File "Prog.py", line 5, in <module>
```

```
    print n[1]
```

```
TypeError: 'int' object has no attribute '__getitem__'
```

# Typage statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

# Typage statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa **compilation**, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

# Typage statique

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa **compilation**, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

Cette information est donnée lors de la compilation :

```
Prog.c: In function 'main':
Prog.c:8:25: error: subscripted value is neither array nor pointer nor vector
    printf("%d\n", n[1]);
                    ^
```

## **Typage dynamique.**

- Avantage : grande flexibilité dans l'écriture des programmes.

## **Typage dynamique.**

- **Avantage** : grande flexibilité dans l'écriture des programmes.
- **Inconvénients** : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

# Avantages et inconvénients

## **Typage dynamique.**

- Avantage : grande flexibilité dans l'écriture des programmes.
- Inconvénients : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

## **Typage statique.**

- Avantages : sécurité lors de l'écriture du programme (les erreurs les plus courantes sont détectées à la compilation), bonne efficacité lors de l'exécution.

# Avantages et inconvénients

## **Typage dynamique.**

- Avantage : grande flexibilité dans l'écriture des programmes.
- Inconvénients : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

## **Typage statique.**

- Avantages : sécurité lors de l'écriture du programme (les erreurs les plus courantes sont détectées à la compilation), bonne efficacité lors de l'exécution.
- Inconvénient : moins de flexibilité dans l'écriture des programmes.

# Typage explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

# Typage explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

- 1 **typage explicite**, où les **types** des variables sont **mentionnés** dans le programme ;

# Typage explicite vs implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables utilisées dans un programme :

- 1 **typage explicite**, où les **types** des variables sont **mentionnés** dans le programme ;
- 2 **typage implicite**, où les **types** des variables ne sont **pas mentionnés** dans le programme. Ils sont devinés lors de la compilation (si typage statique) ou lors de l'exécution (si typage dynamique) en fonction du contexte.