Programmation fonctionnelle

Samuele Giraudo

Université Gustave Eiffel

LIGM, bureau 4B162

samuele.giraudo@u-pem.fr

http://igm.univ-mlv.fr/~giraudo/

Prélude n°1

Les arbres binaires de recherche en programmation fonctionnelle

Arbres binaires

Un arbre binaire est

Arbres binaires

Un arbre binaire est

1. soit une feuille


;

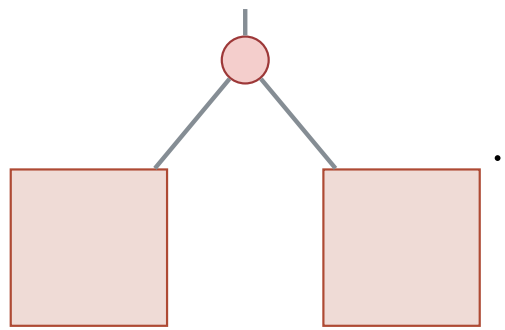
Arbres binaires

Un arbre binaire est

1. soit une feuille

;

2. soit un nœud attaché à deux arbres binaires



.

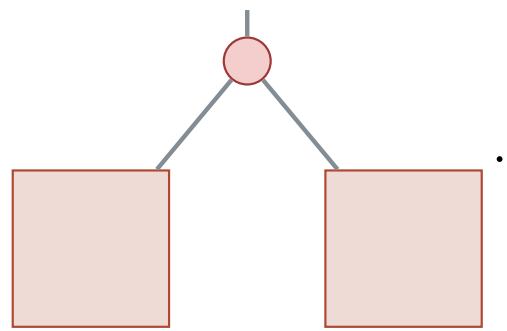
Arbres binaires

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.

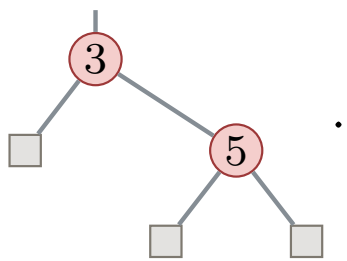
Définition du type arbre binaire en CAML :

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

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

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

Cet arbre se définit de la manière suivante :

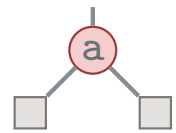
Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

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

Cet arbre se définit de la manière suivante :

1. si $T =$  , alors

$$T_{\leftarrow \mathtt{a}} := $$ 

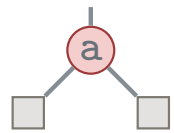
Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

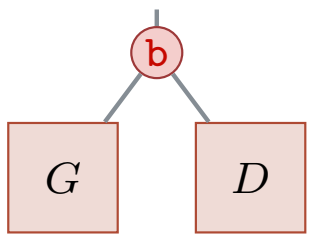
On note $T_{\leftarrow \mathtt{a}}$ l'arbre binaire obtenu après insertion de la lettre a dans T.

Cet arbre se définit de la manière suivante :

1. si $T =$  , alors

$$T_{\leftarrow \mathtt{a}} :=$$ 

2. sinon, T est de la forme

$$T =$$ 

et,

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

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

Cet arbre se définit de la manière suivante :

1. si $T = \square$, alors

$$T_{\leftarrow \mathtt{a}} := \quad \mathtt{a}$$

2. sinon, T est de la forme

$$T = \quad \mathtt{b} \quad G \quad D \qquad \text{et,}$$

- si $\mathtt{a} \leqslant \mathtt{b}$, alors

$$T_{\leftarrow \mathtt{a}} := \quad \mathtt{b} \quad G_{\leftarrow \mathtt{a}} \quad D$$

Insertion dans un arbre binaire de recherche

Soit T arbre binaire de recherche.

On note $T_{\leftarrow \mathtt{a}}$ 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

$$T_{\leftarrow \mathtt{a}} := \quad \begin{array}{c} \mathtt{a} \\ \square \quad \square \end{array}$$

2. sinon, T est de la forme

$$T = \quad \begin{array}{c} \mathtt{b} \\ G \quad D \end{array} \quad \text{et,}$$

- si $\mathtt{a} \leqslant \mathtt{b}$, alors

$$T_{\leftarrow \mathtt{a}} := \quad \begin{array}{c} \mathtt{b} \\ G_{\leftarrow \mathtt{a}} \quad D \end{array}$$

- sinon $\mathtt{a} > \mathtt{b}$ et

$$T_{\leftarrow \mathtt{a}} := \quad \begin{array}{c} \mathtt{b} \\ G \quad D_{\leftarrow \mathtt{a}} \end{array} .$$

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 (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)
```

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
```

Prélude n°2

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 (`range`, `append`, *etc.*).

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 :

$$\texttt{f([0; 1; 2; 3; 4])} \rightarrow \texttt{0 :: f([2; 3; 4])} \rightarrow \texttt{0 :: 2 :: f([4])} \rightarrow \texttt{0 :: 2 :: [4]} \rightsquigarrow \texttt{[0; 2; 4]}$$

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 :

$$\texttt{f([0; 1; 2; 3; 4])} \to \texttt{0 :: f([2; 3; 4])} \to \texttt{0 :: 2 :: f([4])} \to \texttt{0 :: 2 :: [4]} \rightsquigarrow \texttt{[0; 2; 4]}$$

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 **réécrivant** l'expression tant que possible. La définition de fonctions permet d'expliquer comment réaliser les réécritures.

Prélude n°3

Principes généraux

Quelques caractéristiques du CAML

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

Quelques caractéristiques du CAML

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

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

Quelques caractéristiques du CAML

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

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

Quelques caractéristiques du CAML

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

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

  $\rightsquigarrow$ 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**.

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

  $\rightsquigarrow$ 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**.

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

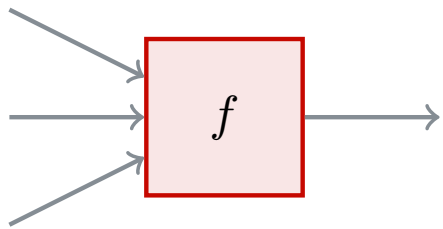
  $\rightsquigarrow$ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

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

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

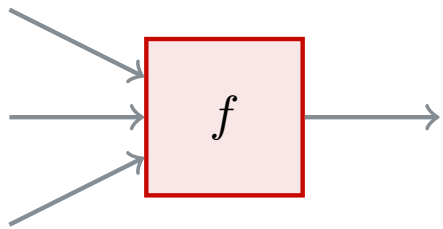
Principes de la programmation fonctionnelle

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

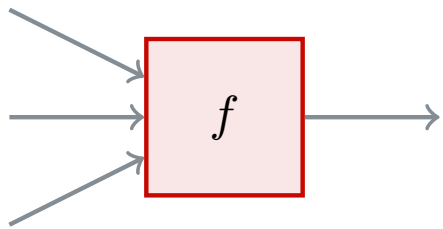
La programmation fonctionnelle est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



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** :

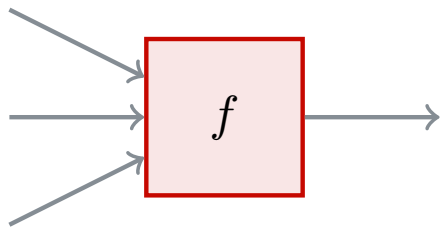


On **évite les effets de bord** (et donc, on ne fait pas d'affectation).

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 **récursivité**.

Pré-requis

Ce cours demande les pré-réquis suivants :

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

Pré-requis

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) ;

Pré-requis

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 ;

Pré-requis

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 1re 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 1re 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 1re 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 1re 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, λ-calcul.

Contenu du cours

Axe 1.

Théorie

Caractéristiques

Axe 2.

Programmation

Pratique

Types

Axe 3.

Notions

Listes

λ-calcul

Bibliographie

Bibliographie non exhaustive :

- X. Leroy, P. Weis, *Le langage Caml*, Dunod, 2e 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.
- X. Leroy *et al.*, The OCaml system release 4.09, 2019.
  Lien : `http://caml.inria.fr/pub/docs/manual-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

Théorie

Caractéristiques

Plan

Théorie

- Machines de Turing
- Décidabilité et indécidabilité
- λ-calcul

Plan

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.
- 1936 : Turing invente la machine de Turing.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.
- 1936 : Turing invente la machine de Turing.
- Années 1940 : programmation en assembleur et en langage machine.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.
- 1936 : Turing invente la machine de Turing.
- Années 1940 : programmation en assembleur et en langage machine.
- Années 1950-1960 : 1ers vrais langages de programmation : FORTRAN, COBOL et Lisp.

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le métier à tisser de Jackard.
- 1936 : Church introduit le λ-calcul.
- 1936 : Turing invente la machine de Turing.
- Années 1940 : programmation en assembleur et en langage machine.
- Années 1950-1960 : 1ers 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, Δ) où

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, Δ) où

1. E est un ensemble fini d'états ;

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, Δ) où

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

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, Δ) 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 ;

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, Δ) 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. $\Delta : E \times \{\cdot, 0, 1\} \to E \times \{\cdot, 0, 1\} \times \{\mathrm{G}, \mathrm{D}\}$ 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	$\cdots$	$u_{\lvert u\rvert}$	$\cdot$	$\cdot$	$\cdots$

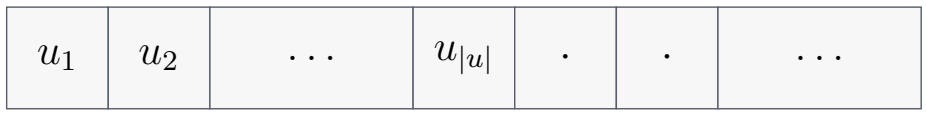
   Les cases à droite de u sont remplies jusqu'à l'infini de $\cdot$.

Fonctionnement des machines de Turing

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

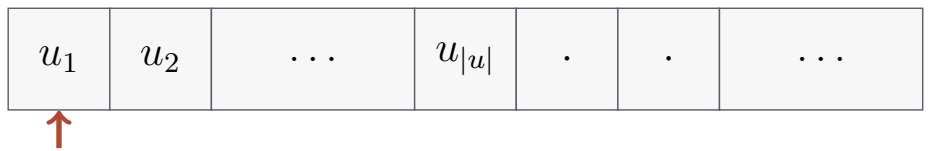
L'exécution de M sur u consiste à :

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 $\cdot$.

2. Placer la **tête de lecture / écriture** sur la 1re case du ruban :



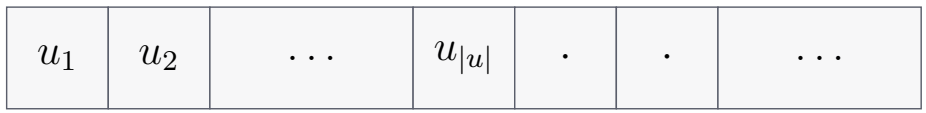
   On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture / écriture à un instant donné.

Fonctionnement des machines de Turing

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

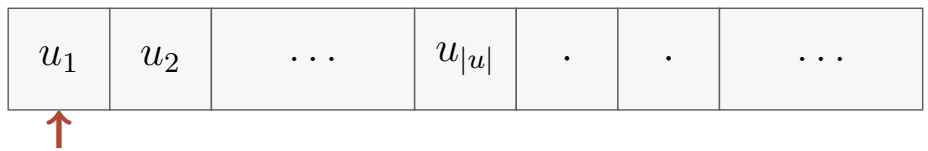
L'exécution de M sur u consiste à :

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 $\cdot$.

2. Placer la **tête de lecture / écriture** sur la 1re case du ruban :



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

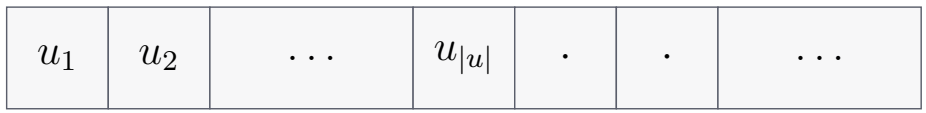
3. Affecter au **registre d'état** e la valeur i (l'état initial).

Fonctionnement des machines de Turing

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

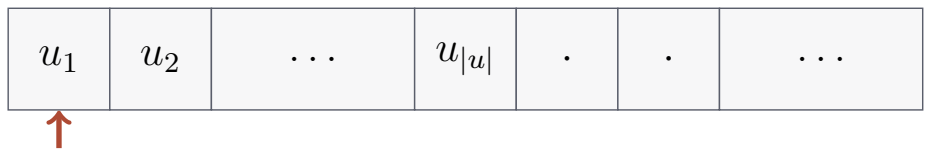
L'exécution de M sur u consiste à :

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 $\cdot$.

2. Placer la **tête de lecture / écriture** sur la 1re 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 (l'état initial).

4. Réaliser les actions dictées par Δ, 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 $s' = \mathrm{D}$, 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 $s' = \mathrm{D}$, 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 $s' = \mathrm{D}$, 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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Exemple : complémentaire d'un mot

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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Calcul de $M(00101)$:

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

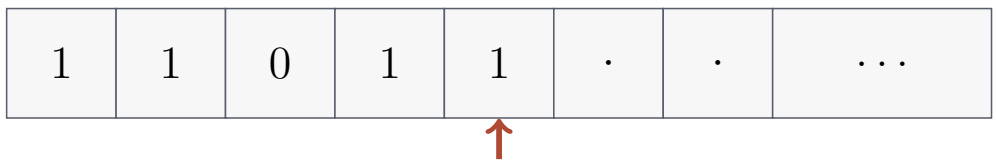
Exemple : complémentaire d'un mot

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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Calcul de $M(00101)$:

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



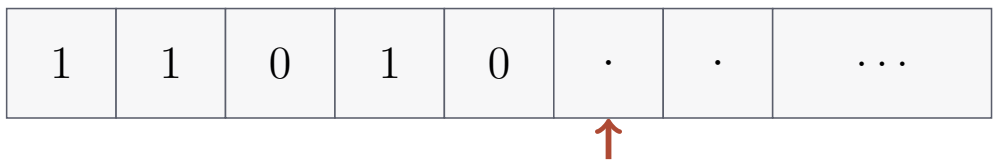
Exemple : complémentaire d'un mot

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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Calcul de $M(00101)$:

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



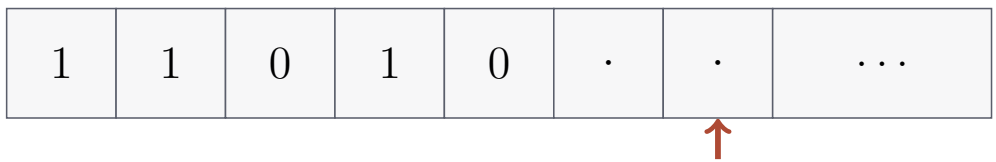
Exemple : complémentaire d'un mot

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

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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 Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

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.

Ici, ce programme Δ 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 Δ est défini par

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

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 Δ est défini par

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

Calcul de $M(001)$:

Exécutions qui ne se terminent pas

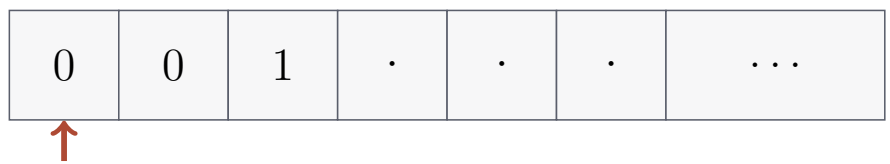
Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u ne se termine pas.

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

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

Calcul de $M(001)$:

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



Exécutions qui ne se terminent pas

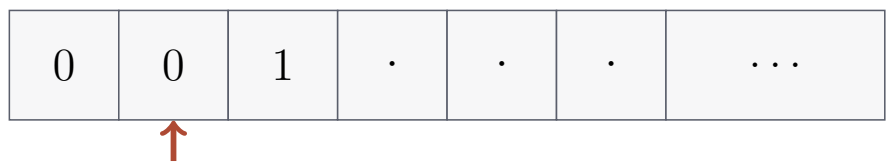
Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u ne se termine pas.

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

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

Calcul de $M(001)$:

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



Exécutions qui ne se terminent pas

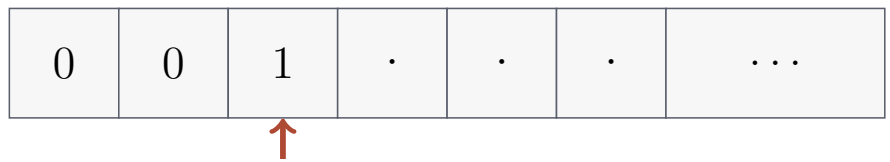
Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u ne se termine pas.

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

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

Calcul de $M(001)$:

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



Exécutions qui ne se terminent pas

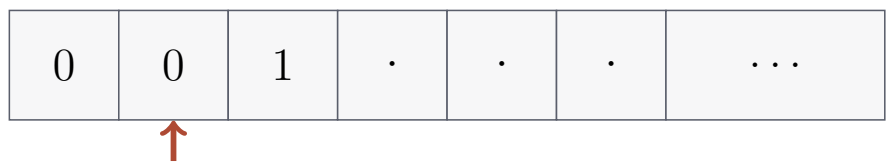
Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u ne se termine pas.

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

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

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 Δ est défini par

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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 Δ est défini par

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

Calcul de $M(001)$:

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

0	0	1	$\cdot$	$\cdot$	$\cdot$	$\cdots$

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 Δ est défini par

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

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 Δ est défini par

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

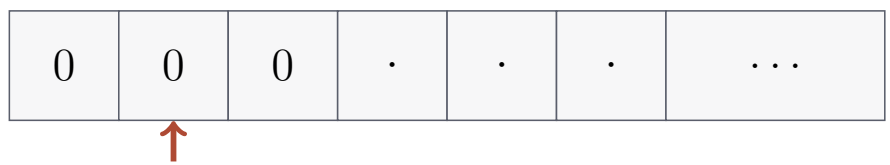
Calcul de $M(001)$:

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

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

Calcul de $M(000)$:

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



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 Δ est défini par

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

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 Δ est défini par

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

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 Δ est défini par

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

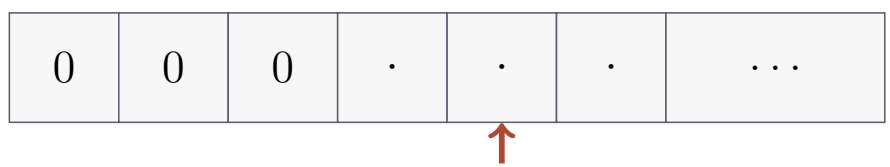
Calcul de $M(001)$:

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

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

Calcul de $M(000)$:

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



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 Δ est défini par

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

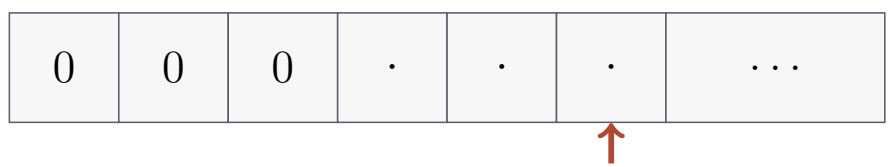
Calcul de $M(001)$:

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

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

Calcul de $M(000)$:

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



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 Δ est défini par

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

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 $\mathrm{code}(M)$ de la manière suivante :

Coder une machine de Turing par un entier naturel

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

1. on fixe des conventions (arbitraires mais rigoureuses) 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) 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 $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) 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 $\mathrm{code}(M)$ 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 Δ est défini par

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

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 Δ est défini par

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

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 Δ est défini par

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

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

$$u = \texttt{0110010101110100}\ldots\texttt{00101001}$$

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

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 Δ est défini par

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

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

$$u = 0110010101110100\ldots00101001$$

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

En décimal :

$$\mathrm{code}(M) = 119527348352246394769818842856657399486293905629080176290359813575714029487388100550026061043720795740337226979189145773772473673716516841910132913242275141863782405111884164058543999074736181406429955364904425229975166545029466892832412634123268241667345353999388604458150336894449185720558624721864880882898175696 9.$$

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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 $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

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

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ 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.

Plan

Problèmes de décision

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 ? $P(010010) = \mathrm{oui}$, $P(011) = \mathrm{non}$;

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

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

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

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

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

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.

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

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

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 Arr est indécidable par l'absurde en supposant que Arr est décidable.

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

Soit l'entier positif $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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, $\mathrm{absurde}$ est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que $\mathrm{absurde}$.

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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 $\mathrm{absurde}$ se termine.

Ainsi, la valeur de retour d'$\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 $\mathrm{absurde}$ se termine.

Ainsi, la valeur de retour d'$\mathrm{absurde}$ figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par $\mathrm{absurde}$ 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 $\mathrm{absurde}$,

  (a) si $M_{\mathrm{Arr}}(u) = 1$, 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 $\mathrm{absurde}$ se termine.

Ainsi, la valeur de retour d'$\mathrm{absurde}$ figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par $\mathrm{absurde}$ ne figure pas dans E (étape (3)).

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