

B2

COMPILATION & ASSEMBLEUR

PLAN

- Introduction à la compilation et à la théorie des langages
- Analyse lexicale et syntaxique
- Grammaire attribuée et arbre syntaxique
- Table de symboles, typage et analyse sémantique
- Représentation Intermédiaire et génération de code
- Allocation de registres

COMPILATION

Introduction à la compilation et TAL

INTRODUCTION

- Compilation :
 - Entrée : un algorithme implémenté sous forme d' un programme dans un langage de programmation « évolué » (de haut niveau)

COMPILATEUR
 - Sortie : un programme équivalent exécutable par un processeur
 - Exemple : C avec gcc

INTRODUCTION

- Interprétation

- Entrée : un algorithme implémenté sous forme d' un programme dans un langage de programmation « évolué » (de haut niveau)

INTERPRETEUR

- Sortie : exécution directe
 - Exemple : Bash, Python

INTRODUCTION

- Mélange

- Entrée : un algorithme implémenté sous forme d' un programme dans un langage de programmation « évolué » (de haut niveau)

COMPILATEUR

- Forme intermédiaire : un programme équivalent dans un autre langage « moins évolué »

INTERPRETEUR

- Sortie : exécution de la forme intermédiaire par un programme interpréteur ou machine virtuelle
 - Exemple : JAVA

INTRODUCTION

- De manière générale
 - Entrée : un fichier définissant des objets informatiques (exécution, données, ...) en utilisant un format ou syntaxe ou langage

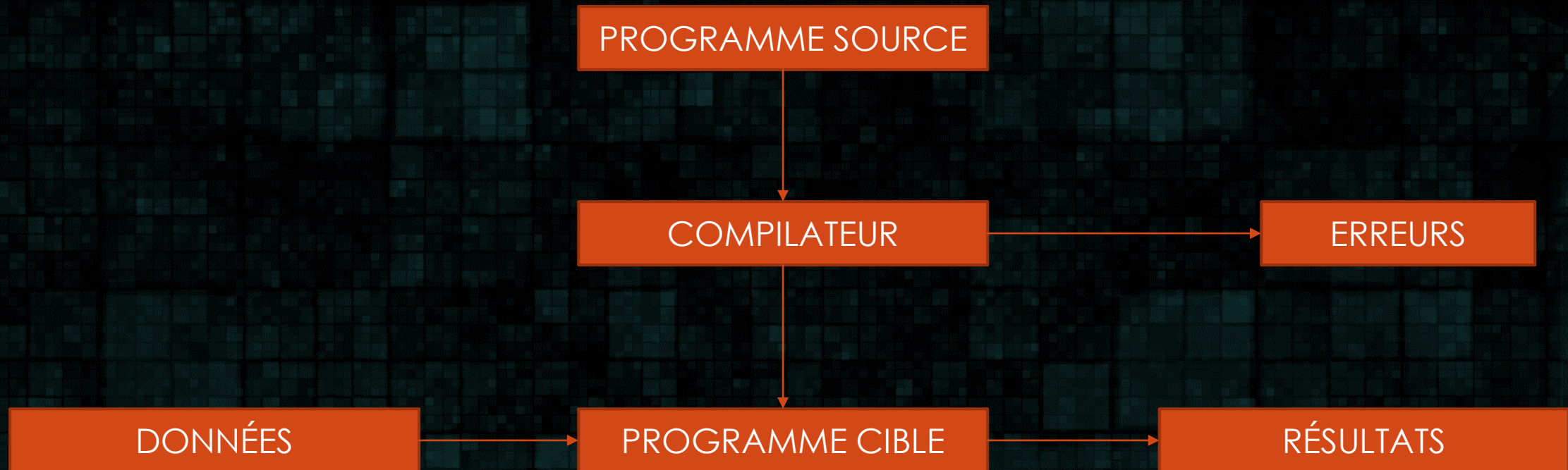
TRADUCTEUR

- Sortie : un fichier équivalent (ou pas) utilisant un autre langage

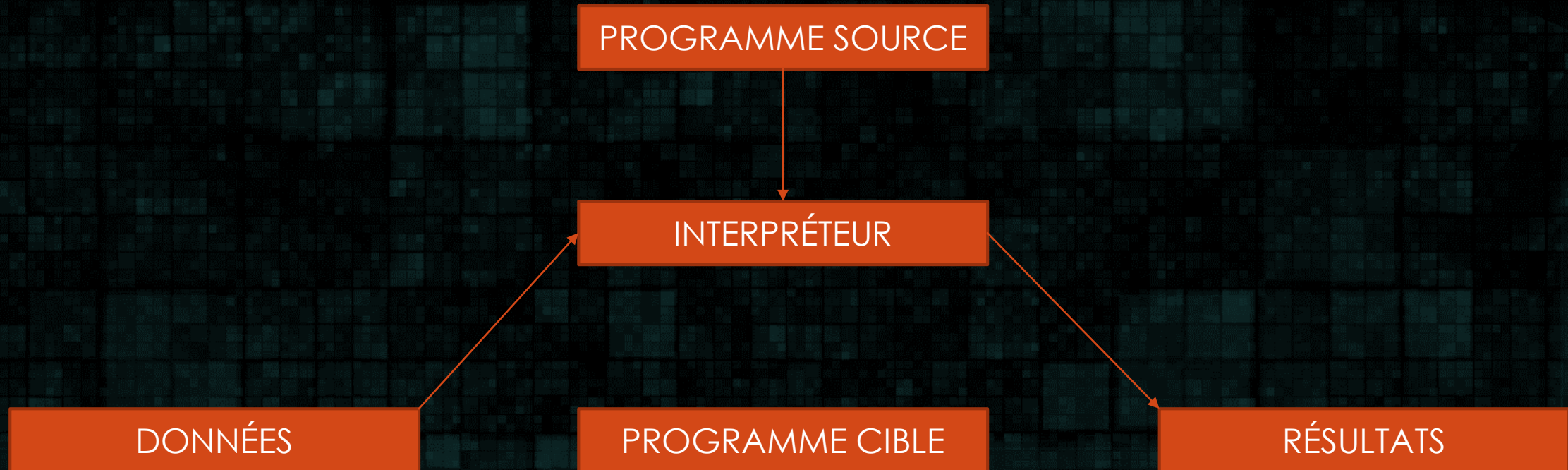
INTRODUCTION

- Un compilateur est un programme
 - qui lit un autre programme rédigé dans un langage de programmation, appelé **langage source**
 - et qui le traduit dans un autre langage, le **langage cible**.
- De plus, le compilateur signale toute erreur contenue dans le programme source

INTRODUCTION



INTRODUCTION



INTRODUCTION

- Exemple :

Programme source

```
entier d;  
  
f(entier a, entier b)  
entier c, entier k;  
{  
    k = a + b;  
    retour k;  
}  
  
main()  
{  
    d = 7;  
    ecrire(f(d, 2) + 1);  
}
```

Programme cible

```
f:  
    push ebp  
    mov  ebp, esp  
    sub  esp, 8  
    mov  ebx, [ebp + 12]  
    push ebx  
    mov  ebx, [ebp + 8]  
    push ebx  
    ...  
main:  
    push ebp  
    mov  ebp, esp  
    sub  esp, 8  
    push 7  
    pop  ebx  
    mov  [d], ebx  
    ...
```


INTRODUCTION

- La traduction ne peut malheureusement pas se faire « mot à mot »
- Le programme source doit être décomposé en **composants pertinents** ou **constructions** du langage source.
- La traduction d'une construction dépend de la **position** qu'elle occupe au sein du programme.

INTRODUCTION

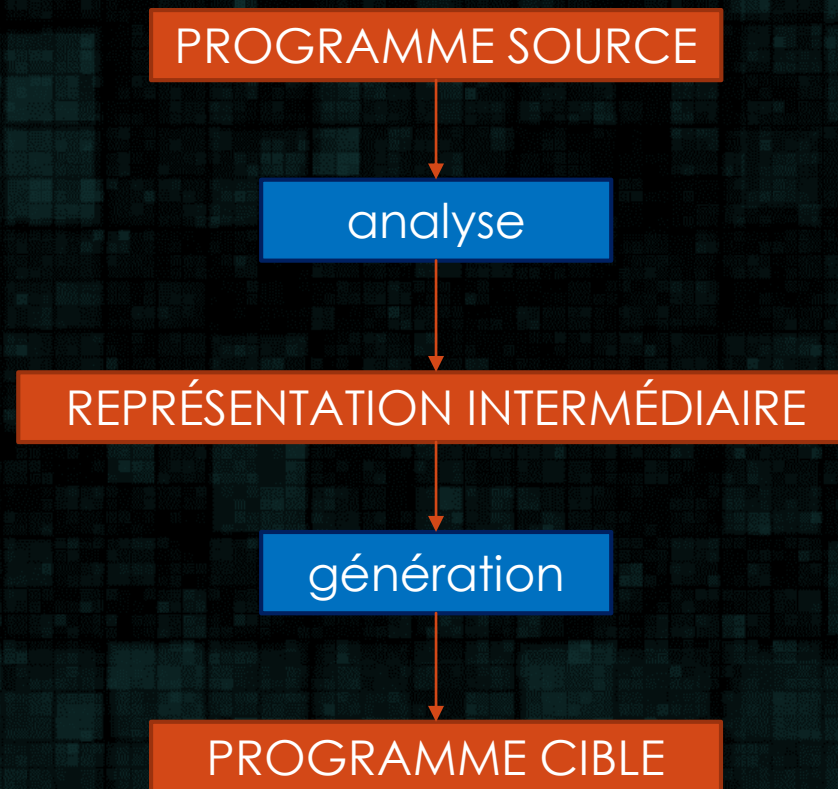
```
entier d;                /* variable globale */

f                          /* nom de la fonction */
(entier a, entier b)      /* paramètres de la fonction */
entier c, entier k;       /* variables locales */
{                          /* debut du corps de la fonction */
    k =                   /* affectation */
        a + b;           /* expression arithmétique */
    retour k;             /* valeur de retour */
}                          /* fin du corps de la fonction */
```

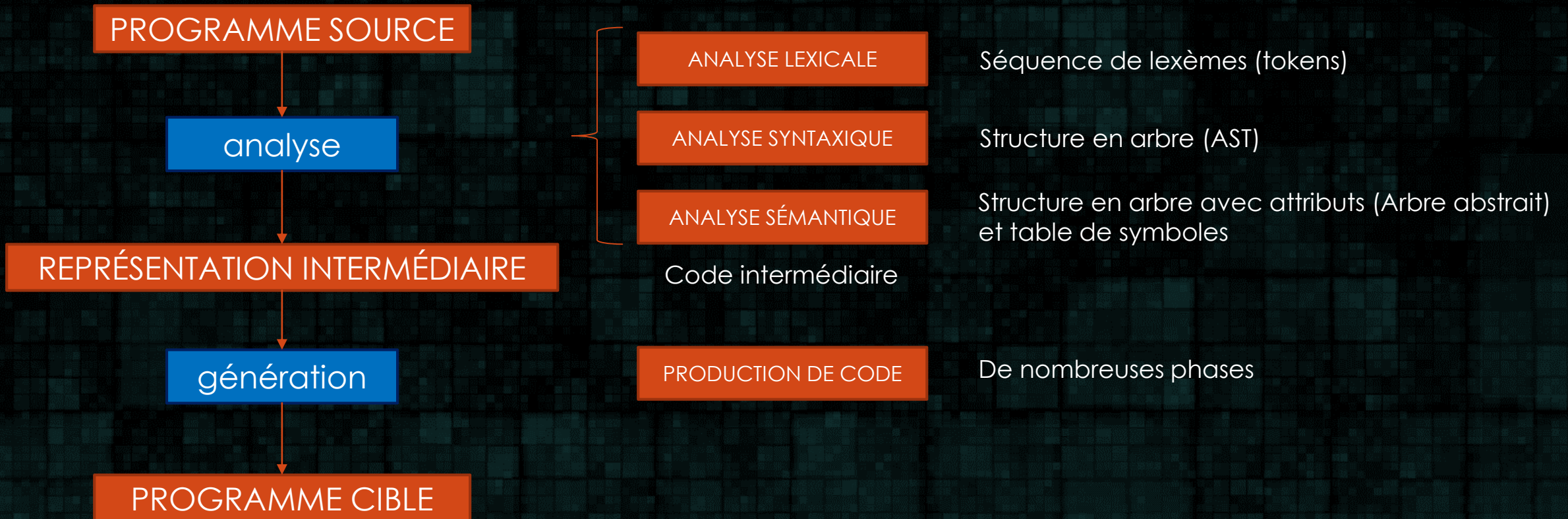

INTRODUCTION

- L'analyse, réalisée par la partie frontale du compilateur, qui :
 - découpe le programme source en ses constituants ;
 - détecte des erreurs de syntaxe ou de sémantique ;
 - produit une représentation intermédiaire du programme source ;
 - conserve dans une table des symboles diverses informations sur les procédures et variables du programme source.
- La génération, réalisée par la partie finale du compilateur, qui :
 - construit le programme cible à partir de la représentation intermédiaire et de la table des symboles

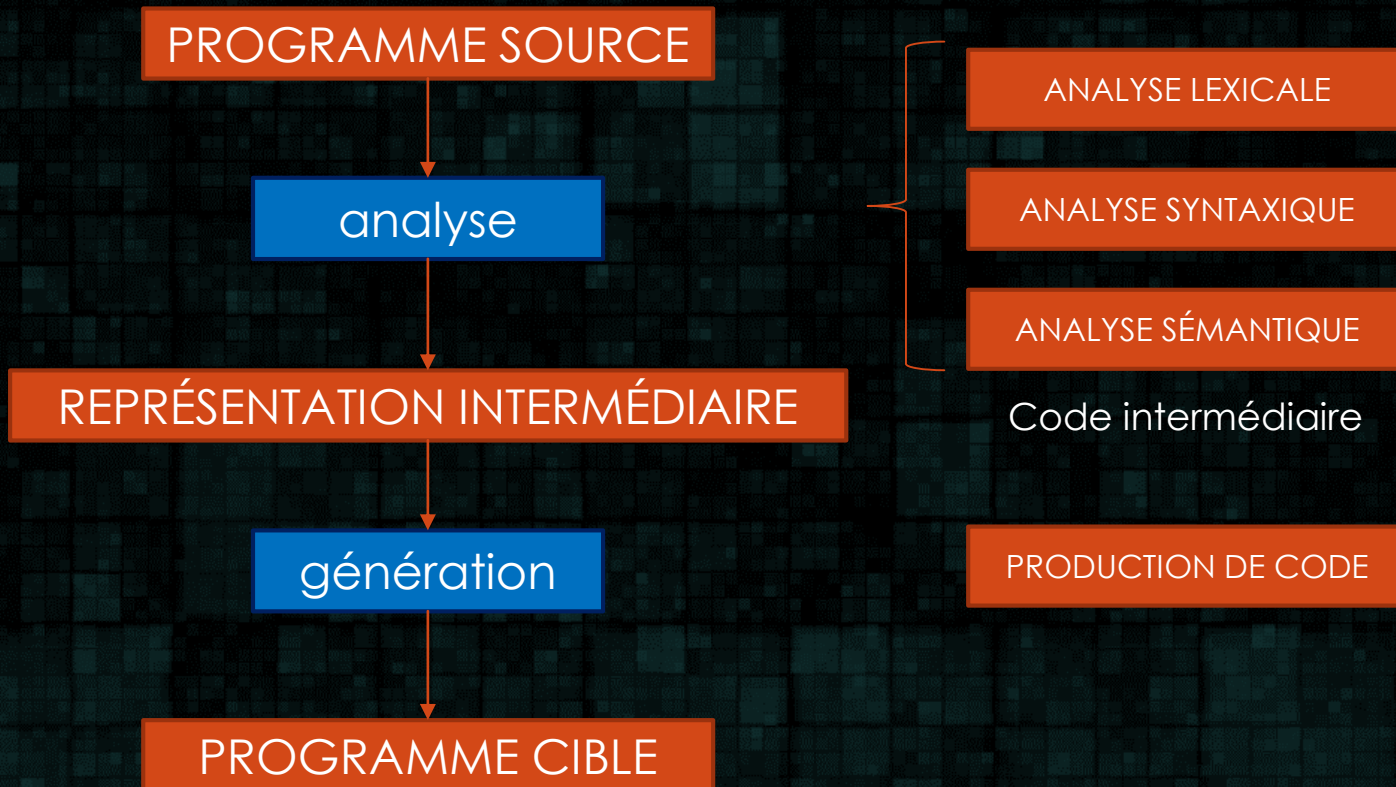
INTRODUCTION



INTRODUCTION

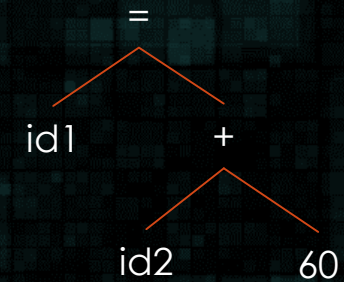


INTRODUCTION



`a = b + 60 ;`

`Id1 = id2 + 60 ;`



```
temp1 = entierVersReel(60)
temp2 = id2 + temp1
Id1 = temp2
mov id2, r2
add 60, r2
mov r2, id1
```



INTRODUCTION

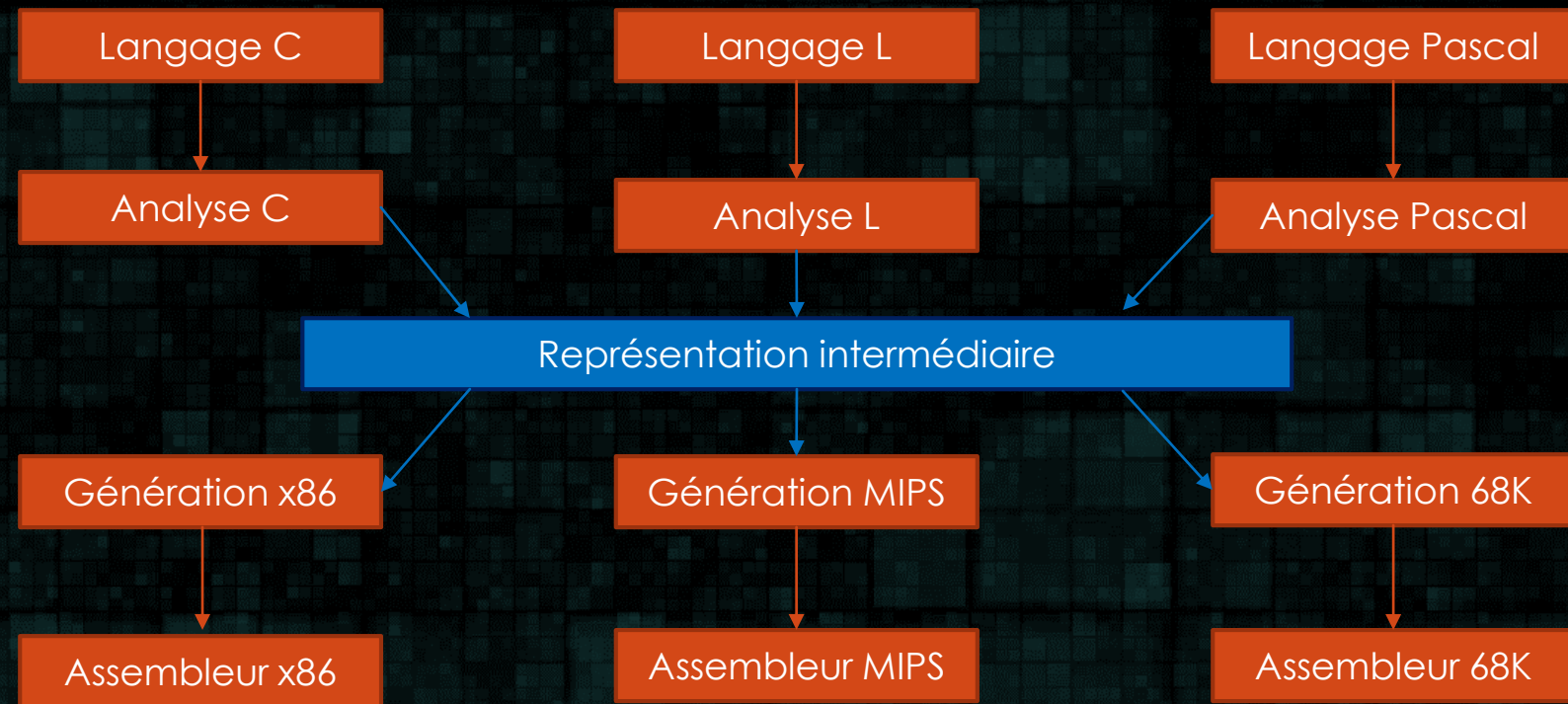
- Nature de la représentation intermédiaire
 - La conception d'une bonne RI est un compromis :
 - Elle doit être raisonnablement facile à produire à partir du programme source.
 - Elle doit être raisonnablement facile à traduire vers le langage cible. Elle doit donc être raisonnablement éloignée (ou raisonnablement proche) du langage source et du langage cible.

INTRODUCTION

- Economie
 - Une RI judicieusement définie permet de construire un compilateur pour le langage L et la machine M en combinant :
 - un analyseur pour le langage L
 - un générateur pour la machine M
 - Economie : on obtient $m \times n$ compilateurs en écrivant seulement m analyseurs et n générateurs

INTRODUCTION

- Portabilité



INTRODUCTION

- Syntaxe du langage source
 - Le programme source vérifie un certain nombre de **contraintes syntaxiques**.
 - L'ensemble de ces contraintes est appelé **grammaire** du langage source.
 - Si le programme ne respecte pas la grammaire du langage, il est considéré incorrect et le processus de compilation échoue.

INTRODUCTION

- Description de la grammaire du langage source
 - Littéraire
 - Un **programme** est une suite de **définitions de fonction**
 - Une **définition de fonction** est composée
 - du **nom de la fonction** suivie de ses **arguments**
 - suivie de la **declaration de ses variables internes**
 - suivie d'un **bloc d'instructions**
 - Une **instruction** est . . .
 - Formelle
 - Programme \rightarrow listeDecFonc '.'
 - listeDecFonc \rightarrow decFonc listeDecFonc
 - listeDecFonc \rightarrow
 - decFonc \rightarrow IDENTIF listeParam listeDecVar ';' instrBloc . . .

INTRODUCTION

- Grammaires formelles

- Les contraintes syntaxiques sont représentées sous la forme de **règles de réécriture**.
- La règle $A \rightarrow BC$ nous dit que le **symbole** A peut se réécrire comme la suite des deux symboles B et C.
- L'ensemble des règles de réécriture constitue la **grammaire** du langage.
- La grammaire d'un langage L permet de générer **tous** les programmes corrects écrits en L et **seulement ceux-ci**

INTRODUCTION

- Notations et Terminologie

- Dans la règle $A \rightarrow \alpha$
 - A est appelé **partie gauche** de la règle.
 - α est appelé **partie droite** de la règle.
- Lorsque plusieurs règles partagent la même partie gauche :
 - $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_n$
- On les note :
 - $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$

INTRODUCTION

- Grammaire partielle des expressions arithmétiques
 - $\text{EXPRESSION} \rightarrow \text{EXPRESSION OP2 EXPRESSION}$
 - $\text{OP2} \rightarrow + \mid - \mid * \mid /$
 - $\text{EXPRESSION} \rightarrow \text{NOMBRE}$
 - $\text{EXPRESSION} \rightarrow (\text{EXPRESSION})$
 - $\text{NOMBRE} \rightarrow \text{CHIFFRE} \mid \text{CHIFFRE NOMBRE}$
 - $\text{CHIFFRE} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- Les **symboles** EXPRESSION, OP2, NOMBRE, CHIFFRE sont appelés **symboles non terminaux** de la grammaire
- Les symboles +, -, *, /, (,), 0, 1, ..., 9 sont appelés **symboles terminaux** de la grammaire

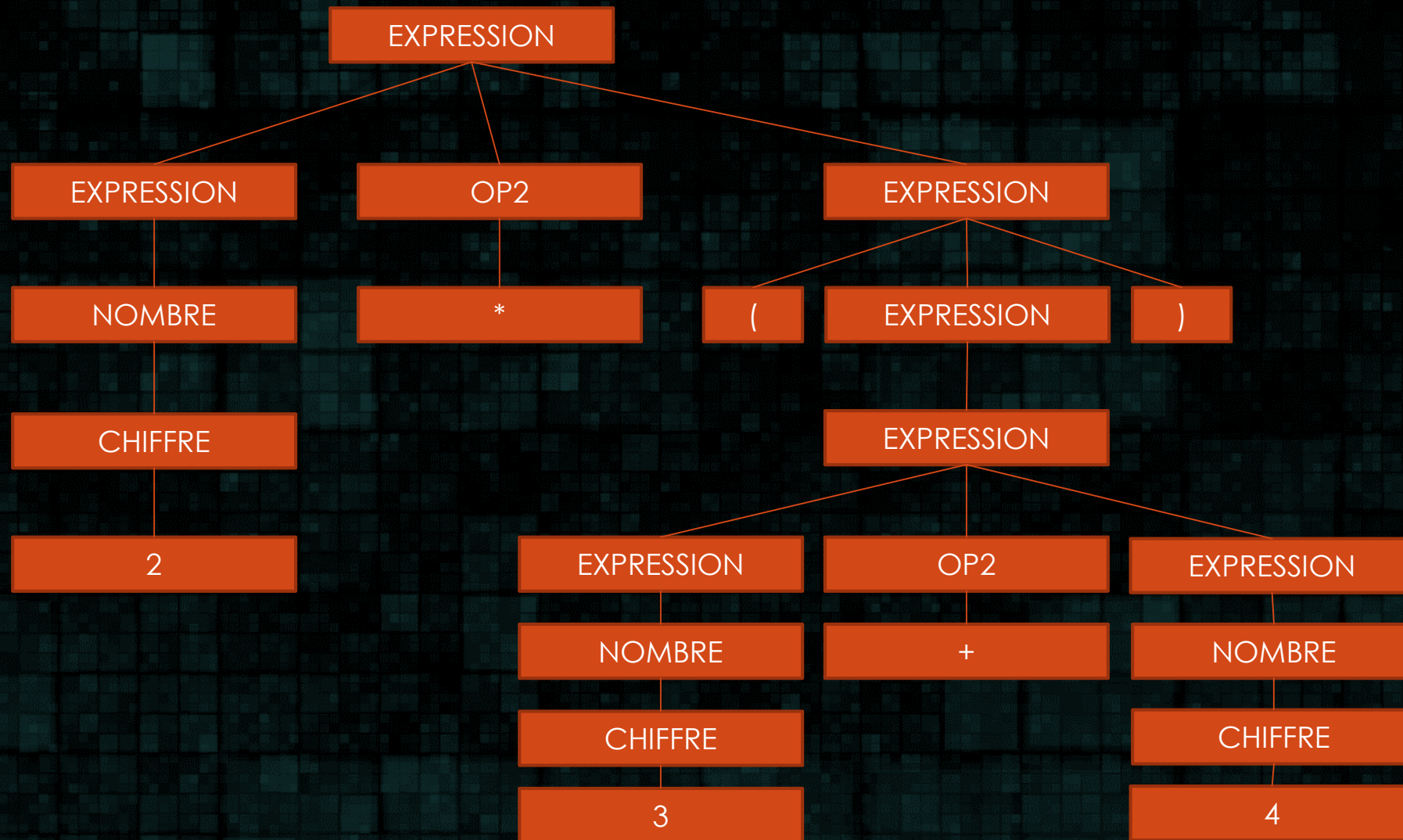
INTRODUCTION

- Avantages des grammaires formelles
 - Une grammaire formelle :
 - Pousse le concepteur d'un langage à en décrire la syntaxe de manière **exhaustive**.
 - Permet de répondre automatiquement à la question mon programme est-il correct ? à l'aide d'un **analyseur syntaxique**.
 - Fournit, à l'issue de l'analyse, une représentation explicite de l'organisation du programme en constructions (**structure syntaxique du programme**).
 - Cette représentation est utile pour la suite du processus de compilation

INTRODUCTION

- Dérivation d'une expression arithmétique
- L'expression arithmétique $2*(3+1)$ est-elle correcte ?

▪	EXPRESSION	⇒	EXPRESSION OP2 EXPRESSION
		⇒	NOMBRE OP2 EXPRESSION
		⇒	CHIFFRE OP2 EXPRESSION
		⇒	2 OP2 EXPRESSION
		⇒	2 * EXPRESSION
		⇒	2 * (EXPRESSION)
		⇒	2 * (EXPRESSION OP2 EXPRESSION)
		⇒	2 * (NOMBRE OP2 EXPRESSION)
		⇒	2 * (CHIFFRE OP2 EXPRESSION)
		⇒	2 * (3 OP2 EXPRESSION)
		⇒	2 * (3 + EXPRESSION)
		⇒	2 * (3 + NOMBRE)
		⇒	2 * (3 + CHIFFRE)
		⇒	2 * (3 + 1)



INTRODUCTION

- Analyse lexicale
 - Afin de simplifier la grammaire décrivant un langage, on omet de cette dernière la génération de certaines parties simples du langage.
 - Ces dernières sont prises en charge par un **analyseur lexical**
 - L'analyseur lexical traite le programme source et fournit le résultat de son traitement à l'analyseur syntaxique.

INTRODUCTION

- Nouvelle grammaire des expressions arithmétiques

Syntaxe

EXPRESSION → EXPRESSION OP2 EXPRESSION
EXPRESSION → NOMBRE
EXPRESSION → (EXPRESSION)

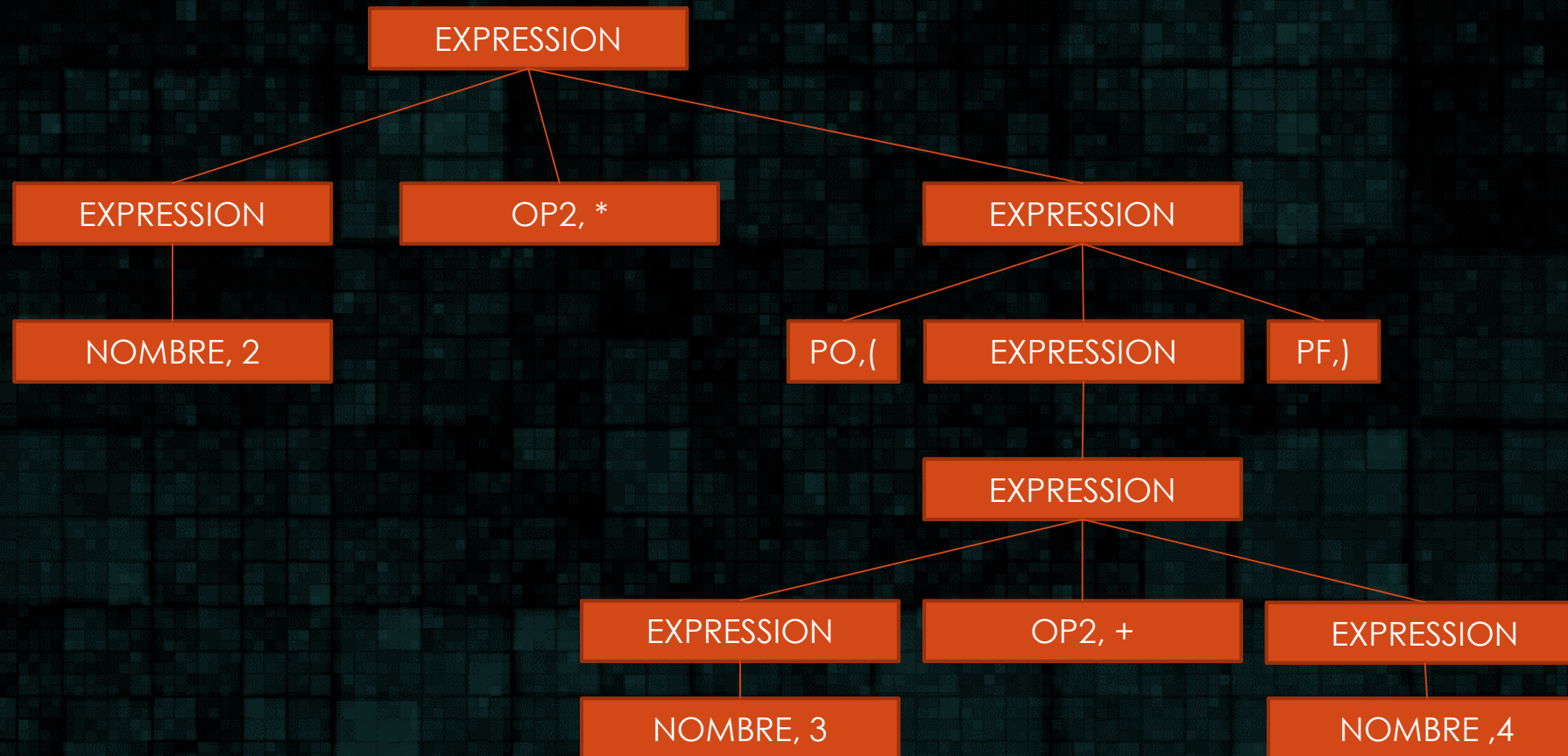
Lexique

OP2→+ - * /
NOMBRE→CHIFFRE CHIFFRE NOMBRE
CHIFFRE→0 1 2 3 4 5 6 7 8 9

INTRODUCTION

- Analyseur lexical
 - Lit le programme source
 - Reconnaît des séquences de caractères significatives appelées **lexèmes**
 - Pour chaque lexème, l'analyseur lexical émet un couple (type du lexème, valeur du lexème)
 - Exemple : (NOMBRE,123)
 - Les types de lexèmes sont des **symboles**, ils constituent les symboles terminaux de la grammaire du langage.
 - Les symboles terminaux de la grammaire (ou types de lexèmes) constituent l'**interface** entre l'analyseur lexical et l'analyseur syntaxique. Ils doivent être connus des deux

INTRODUCTION



TAL ET AUTOMATES

Langages, grammaire et reconnaisseurs

LANGAGES

- Le paysage syntaxique
 - Les **symboles** sont des éléments indivisibles qui vont servir de briques de base pour construire des mots.
 - Un **alphabet** est un ensemble fini de symboles.
 - On désigne conventionnellement un alphabet par la lettre grecque Σ .
 - Une suite de symboles, appartenant à un alphabet Σ , mis bout à bout est appelé un **mot** (ou une chaîne) sur Σ .
 - Le mot de longueur zéro est noté ϵ .
 - On note $|m|$ la longueur du mot m (le nombre de symboles qui le composent) et $|m|_s$ le nombre de symboles s que possède le mot m .
 - L'ensemble de tous les mots que l'on peut construire sur un alphabet Σ est noté Σ^* .
 - Un **langage** sur un alphabet Σ est un ensemble de mots construits sur Σ . Tout langage défini sur Σ est donc une partie de Σ^* .

LANGAGES

- Exemples de langages

- $\Sigma = \{a\}$ $L1 = \{ \varepsilon, a, aa, aaa, \dots \}$
- $\Sigma = \{a,b\}$ $L2 = \{ \varepsilon, ab, aabb, aaabbb, aaaabbbb, \dots \}$
- $\Sigma = \{a,b\}$ $L3 = \{ \varepsilon, aa, bb, aaaa, abba, baab, bbbb, \dots \}$
- $\Sigma = \{a,b,c\}$ $L4 = \{ \varepsilon, abc, aabbcc, aaabbbccc, \dots \}$

LANGAGES

- Opérations sur les langages
 - Union $L1 \cup L2 \{ x | x \in L1 \text{ ou } x \in L2 \}$
 - Intersection $L1 \cap L2 \{ x | x \in L1 \text{ et } x \in L2 \}$
 - Différence $L1 - L2 \{ x | x \in L1 \text{ et } x \notin L2 \}$
 - Complément $\bar{L} \{ x \in \Sigma^* | x \notin L \}$
 - Concaténation $L1L2 \{ xy | x \in L1 \text{ et } y \in L2 \}$
 - Auto concaténation $L \dots L = L^n$
 - Fermeture de Kleene $L^* \cup_{k \geq 0} L^k$

LANGAGES

- Comment décrire un langage ?
 - Énumération
 - $L_2 = \{\epsilon, ab, aabb, aaabbb, aaaabbbb, \dots\}$
 - Description littéraire
 - Ensemble des mots construits sur l'alphabet $\{a,b\}$, commençant par des a et se terminant par des b et tel que le nombre de a et le nombre de b soit égal
 - Grammaire de réécriture
 - $G = \langle \{S\}, \{a, b\}, \{S \rightarrow aSb \mid \epsilon\}, S \rangle$

GRAMMAIRE

- Une grammaire de réécriture est un 4-uplet $\langle N, \Sigma, P, S \rangle$ où :
 - N est un ensemble de symboles non terminaux, appelé l'alphabet non-terminal.
 - Σ est un ensemble de symboles terminaux, appelé l'alphabet terminal, tel que N et Σ soient disjoints.
 - P est un sous ensemble fini de règles de production ou règles de réécriture :
 - $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta$
 - S est un élément de N appelé l'axiome de la grammaire. C'est la désignation d'un des non-terminaux en tant que symbole de départ S

GRAMMAIRE

- Les proto-mots d'une grammaire $G = \langle N, \Sigma, P, S \rangle$ sont des mots construits sur l'alphabet $\Sigma \cup N$, on les définit récursivement de la façon suivante :
 - S est un proto-mot de G
 - si $\alpha\beta\gamma$ est un proto-mot de G et $\beta \rightarrow \delta \in P$ alors $\alpha\delta\gamma$ est un proto-mot de G .
- Un proto-mot de G ne contenant aucun symbole non-terminal est appelé un mot engendré par G .
- Le langage engendré par G , noté $L(G)$ est l'ensemble des mots engendrés par G .

GRAMMAIRE

- Conventions

- Les symboles non terminaux appartenant à N sont représentés par des lettres latines majuscules : A, B, C, S, E, T, \dots
- Les symboles terminaux appartenant à Σ sont représentés par des lettres latines minuscules : a, b, c, d, \dots
- L'axiome est représenté par le non-terminal S et constitue la partie gauche de la première règle de production
- Les proto-mots appartenant à $(N \cup \Sigma)^*$ sont représentés par des lettres grecques minuscules : $\alpha, \beta, \gamma, \varepsilon, \dots$

GRAMMAIRE

- $L1 = \{ \epsilon, a, aa, aaa, \dots \}$

$G = \langle \{S\}, \{a\}, \{S \rightarrow Sa \mid \epsilon\}, S \rangle$

sous-ensemble des proto-mots de G



GRAMMAIRE

- $L2 = \{ \varepsilon, ab, aabb, aaabbb, aaaabbbb \dots \}$

$G = \langle \{ S \}, \{ a, b \}, \{ S \rightarrow aSb \mid \varepsilon \}, S \rangle$

sous-ensemble des proto-mots de G

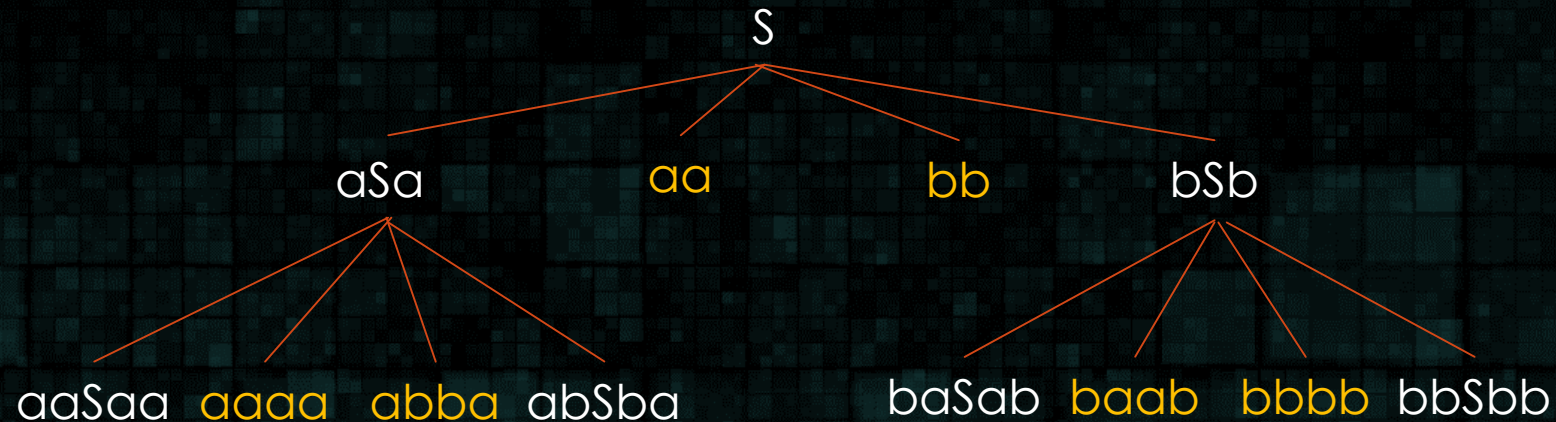


GRAMMAIRE

- $L3 = \{ aa, bb, aaaa, abba, baab, bbbb, \dots \}$

$G = \langle \{ S \}, \{ a, b \}, \{ S \rightarrow aSa \mid bSb \mid aa \mid bb \}, S \rangle$

sous-ensemble des proto-mots de G

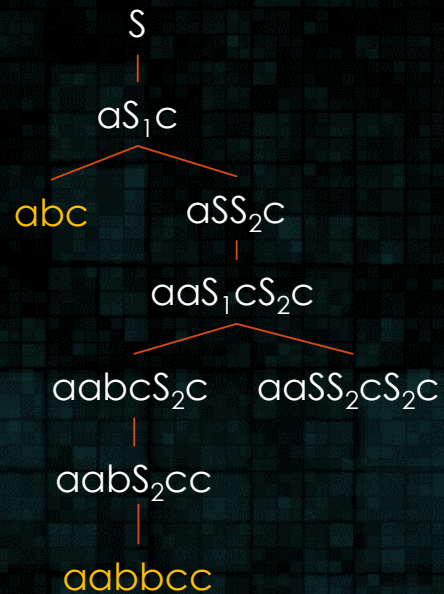


GRAMMAIRE

- $L_4 = \{ \epsilon, abc, aabbcc, aaabbbccc, aaaabbbbccccc \dots \}$

$G = \langle \{ S, S_1, S_2 \}, \{ a, b, c \}, \{ S \rightarrow aS_1c, S_1 \rightarrow b \mid SS_2, cS_2 \rightarrow S_2c, bS_2 \rightarrow bb \}, S \rangle$

sous-ensemble des proto-mots de G



GRAMMAIRE

- Comme vu dans le premier cours, Les proto-mots engendrés lors d'une dérivation peuvent comporter plus d'un symbole non-terminal :

$G = \langle \{E, T, F\}, \{+, *, a\}, \{E \rightarrow T + E \mid T, T \rightarrow F * T \mid F, F \rightarrow a\}, E \rangle$

$E \Rightarrow T + E$
 $\Rightarrow T + T$
 $\Rightarrow F + T$
 $\Rightarrow F + F * T$
 $\Rightarrow F + a * T$
 $\Rightarrow F + a * F$
 $\Rightarrow a + a * F$
 $\Rightarrow a + a * a$

GRAMMAIRE

- Il existe deux sens de dérivation :

Dérivation gauche : on réécrit le non-terminal le plus à gauche :

$$\begin{aligned} E &\Rightarrow T + E \\ &\Rightarrow F + E \\ &\Rightarrow a + E \\ &\Rightarrow a + T \\ &\Rightarrow a + F * T \\ &\Rightarrow a + a * T \\ &\Rightarrow a + a * F \\ &\Rightarrow a + a * a \end{aligned}$$

Dérivation droite : on réécrit le non-terminal le plus à droite :

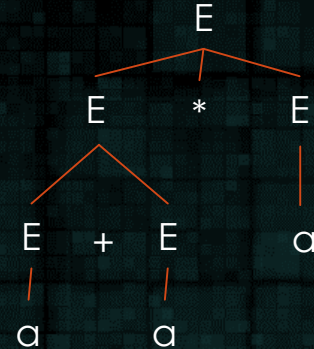
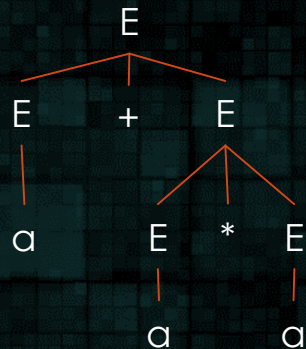
$$\begin{aligned} E &\Rightarrow T + E \\ &\Rightarrow T + T \\ &\Rightarrow F + T \\ &\Rightarrow F + F * T \\ &\Rightarrow F + a * T \\ &\Rightarrow F + a * F \\ &\Rightarrow a + a * F \\ &\Rightarrow a + a * a \end{aligned}$$

GRAMMAIRE

- Arbre de dérivation
 - Un arbre de dérivation indique les règles qui ont été utilisées dans une dérivation, mais pas l'ordre dans lequel elles ont été utilisées.
 - À un arbre de dérivation correspondent une seule dérivation droite et une seule dérivation gauche.

GRAMMAIRE

- Une grammaire G est dite **ambiguë** s'il existe au moins un mot m dans $L(G)$ auquel correspond plus d'un arbre de dérivation.
 - Exemple : $E \rightarrow E + E \mid E * E \mid a$



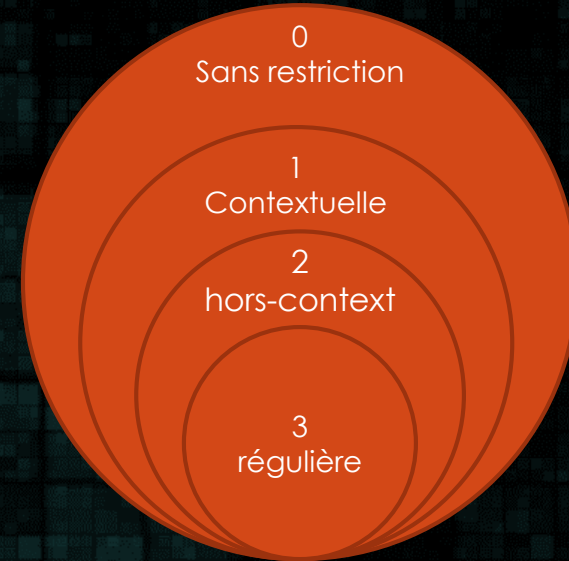
GRAMMAIRE

- Symboles et règles de production utiles
 - Pour manipuler une grammaire, il est souhaitable que tous les symboles et règles de production soient utiles :
 - Un symbole terminal ou non-terminal est utile s'il apparaît dans une règle de production utile
 - Une règle de production est utile si :
 - elle peut générer des mots
 - le symbole non-terminal de la partie gauche peut être généré (sauf l'axiome, qui peut par définition ne pas être généré)
 - elle n'est pas de la forme $\alpha \rightarrow \alpha$

GRAMMAIRE

- Types de règles
 - Les grammaires peuvent être classées en fonction de la forme de leurs règles de production.
 - On définit cinq types de règles de production :
 - Une règle est régulière à gauche si et seulement si elle est de la forme $A \rightarrow xB$ ou $A \rightarrow x$ avec $A, B \in N$ et $x \in \Sigma^*$.
 - Une règle est régulière à droite si et seulement si elle est de la forme $A \rightarrow Bx$ ou $A \rightarrow x$ avec $A, B \in N$ et $x \in \Sigma^*$.
 - Une règle $A \rightarrow \alpha$ est une règle hors-contexte si et seulement si : $A \in N$ et $\alpha \in (N \cup \Sigma)^*$
 - Une règle $\alpha \rightarrow \beta$ est une règle contextuelle si et seulement si : $\alpha = gAd$ et $\beta = gBd$ avec $g, d, B \in (N \cup \Sigma)^*$ et $A \in N$.
Le nom “contextuelle” provient du fait que A se réécrit B uniquement dans le contexte g_d .
 - Une règle $\alpha \rightarrow \beta$ est une règle sans restriction si elle n’est pas contextuelle.

GRAMMAIRE



- Type d'une grammaire

- Une grammaire est :

- Régulière ou de type 3 si elle est régulière à droite ou régulière à gauche. Une grammaire est régulière à gauche si toutes ses règles sont régulières à gauche et une grammaire est régulière à droite si toutes ses règles sont régulières à droite.
 - Hors-contexte ou de type 2 si toutes ses règles de production sont hors-contexte.
 - Dépendante du contexte ou de type 1 si toutes ses règles de production sont dépendantes du contexte.
 - Sans restrictions ou de type 0 si toutes ses règles de production sont sans restrictions.

GRAMMAIRE

- Exemples de langages réguliers :

- $L1 = \{ m \in \{a, b\}^* \}$

- $G1 = \langle \{S\}, \{a, b\}, \{S \rightarrow aS \mid bS \mid \varepsilon\}, S \rangle$

- $L2 = \{ m \in \{a, b\}^* \mid |m|_a \bmod 2 = 0 \}$

- $G2 = \langle \{S, T\}, \{a, b\}, \{S \rightarrow aT \mid bS \mid \varepsilon, T \rightarrow aS \mid bT\}, S \rangle$

- $L3 = \{ m \in \{a, b\}^* \mid m = xaaa \text{ avec } x \in \{a, b\}^* \}$

- $G3 = \langle \{S, T, U\}, \{a, b\}, \{S \rightarrow aS \mid bS \mid aT, T \rightarrow aU, U \rightarrow a\}, S \rangle$

GRAMMAIRE

- Exemples de langages hors-contexte
 - $L1 = \{ a^n b^n \mid n \geq 0 \}$
 - $G1 = \langle \{ S \}, \{ a, b \}, \{ S \rightarrow aSb \mid \epsilon \}, S \rangle$
 - $L2 = \{ mm^{-1} \mid m \in \{ a, b \}^* \}$ (langage miroir - palindromes paires)
 - $G2 = \langle \{ S \}, \{ a, b \}, \{ S \rightarrow aSa \mid bSb \mid \epsilon \}, S \rangle$

GRAMMAIRE

- Exemples de langages contextuels

- $L1 = \{ a^n b^n c^n \mid n \geq 0 \}$

- $G1 = \{ S, \bar{B}, B, C \}, \{ a, b, c \}, S \rightarrow aSBC \mid \varepsilon, CB \rightarrow \bar{B}B, \bar{B}B \rightarrow \bar{B}C, \bar{B}C \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc, S \rangle$

RECONNAISSEUR - AUTOMATE

- Une grammaire d'un langage L permet de générer tous les mots appartenant à L .
- Un reconnaisseur pour un langage L est un programme qui prend en entrée un mot m et répond oui si m appartient à L et non sinon.
- Pour chaque classe de grammaire, il existe une classe de reconnaisseurs qui définit la même classe de langages

Type de grammaire	Type de reconnaisseur
Régulière	Automate fini
Hors-contexte	Automate à pile
Contextuelle	Automate linéairement borné
Sans restriction	Machine de Turing

RECONNAISSEUR - AUTOMATE

- Un reconnaisseur est composé de quatre parties :
 - une **bande de lecture**
 - Elle est composée d'une succession de cases.
 - Chaque case peut contenir un seul symbole d'un alphabet d'entrée.
 - C'est dans les cases de cette bande de lecture qu'est écrit le mot à reconnaître.
 - une **tête de lecture**
 - Elle peut lire une case à un instant donné.
 - La case sur laquelle se trouve la tête de lecture à un moment donné s'appelle la **case courante**.
 - La tête peut être déplacée par le reconnaisseur pour se positionner sur la case immédiatement à gauche ou à droite de la case courante.
 - une **mémoire**
 - Elle peut prendre des formes différentes.
 - La mémoire permet de stocker des éléments d'un **alphabet de mémoire**.

RECONNAISSEUR - AUTOMATE

- une **unité de contrôle**
 - Elle constitue le cœur d'un reconnaisseur.
 - Elle peut être vue comme un programme qui dicte au reconnaisseur son comportement.
 - Elle est définie par un ensemble fini d'**états** ainsi que par une **fonction de transition** qui décrit le passage d'un état à un autre en fonction du contenu de la case courante de la bande de lecture et du contenu de la mémoire.
 - L'unité de contrôle décide aussi de la direction dans laquelle déplacer la tête de lecture et choisit quels symboles stocker dans la mémoire.
 - Parmi les états d'un reconnaisseur, on distingue
 - **des états initiaux**, qui sont les états dans lesquels doit se trouver le reconnaisseur avant de commencer à reconnaître un mot
 - **des états d'acceptation** qui sont les états dans lequel doit se trouver le reconnaisseur après avoir reconnu un mot.

RECONNAISSEUR - AUTOMATE

- Configuration et mouvement
 - Configuration d'un reconnaisseur :
 - Etat de l'unité de contrôle
 - Contenu de la bande d'entrée et position de la tête
 - Contenu de la mémoire
 - Mouvement :
 - passage d'une configuration à une autre ($C_1 \vdash C_2$)

RECONNAISSEUR - AUTOMATE

- Configurations
 - configuration initiale
 - L'unité de contrôle est dans un état initial
 - La tête est au début de la bande
 - La mémoire contient un élément initial.
 - configuration d'acceptation
 - L'unité de contrôle est dans un état d'acceptation
 - La tête de lecture est à la fin de la bande
 - La mémoire se trouve dans un état d'acceptation

RECONNAISSEUR - AUTOMATE

- Déterminisme
 - L'unité de contrôle est dite déterministe si à toute configuration correspond au plus un mouvement pour une transition donnée
 - S'il peut exister plus d'un mouvement, elle est dite non déterministe.

RECONNAISSEUR - AUTOMATE

- Reconnaissance

- Un mot m est acceptée par un reconnaisseur si, partant de l'état initial, avec m sur la bande d'entrée, le reconnaisseur peut faire une série de mouvements pour se retrouver dans un état d'acceptation.
- Le langage accepté par un reconnaisseur est l'ensemble de tous les mots qu'il accepte.

RECONNAISSEUR - AUTOMATE

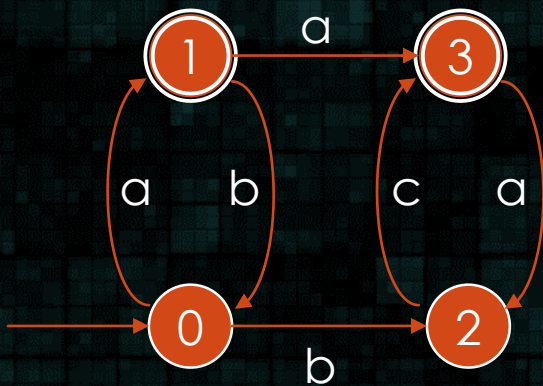
- **Rapports avec la compilation**
 - **Analyse lexicale** → automates finis
 - **Analyse syntaxique** → automates à pile
 - **Production de code** → machines de Turing

AUTOMATES FINIS

- Définition
 - Un automate fini est un 5-uplet $\langle Q, \Sigma, \delta, q_0, F \rangle$
 - Q est l'ensemble des états,
 - Σ est l'alphabet de l'entrée
 - δ est la fonction de transition
 - $q_0 \in Q$ est l'état initial,
 - $F \subseteq Q$ est l'ensemble des états d'acceptation

AUTOMATES FINIS

- Exemple :



$\langle Q, \Sigma, \delta, q_0, F \rangle$

$Q = \{ 0, 1, 2, 3 \}$

$\Sigma = \{ a, b, c \}$

$\delta(0, a) = \{ 1 \}$

$\delta(0, b) = \{ 2 \}$

$\delta(1, a) = \{ 3 \}$

$\delta(1, b) = \{ 0 \}$

$\delta(2, c) = \{ 3 \}$

$\delta(3, a) = \{ 2 \}$

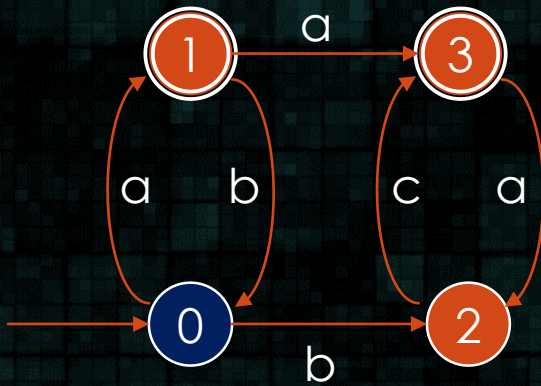
$q_0 = 0$

$F = \{ 1, 3 \}$

AUTOMATES FINIS

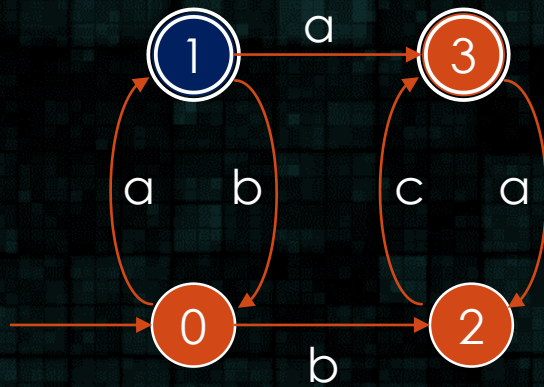
- Reconnaissance :

(0, ababbc)



AUTOMATES FINIS

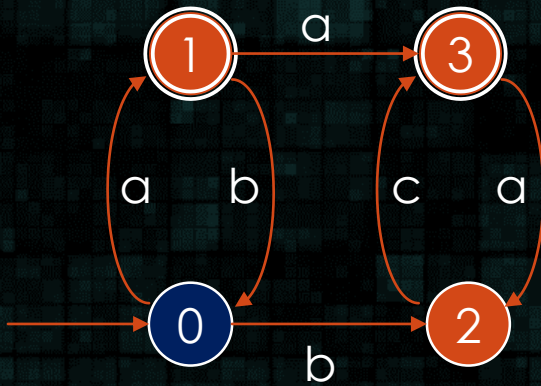
- Reconnaissance :



⊢ (0, ababbc)
(1, babbc)

AUTOMATES FINIS

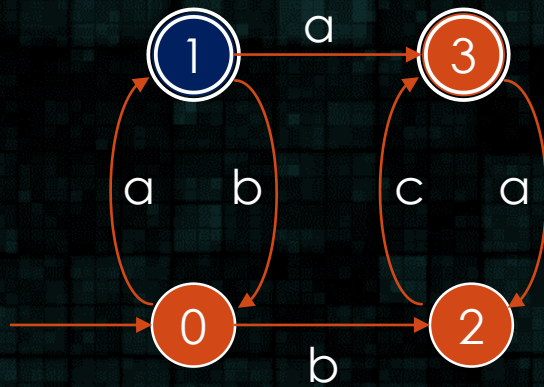
- Reconnaissance :



$\vdash (0, ababbc)$
 $\vdash (1, babbc)$
 $\vdash (0, abbc)$

AUTOMATES FINIS

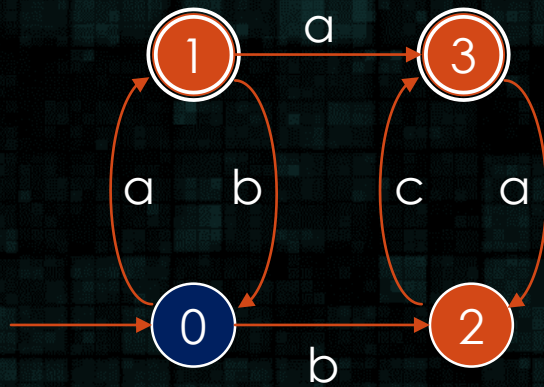
- Reconnaissance :



(0, ababbc)
⊢ (1, babbc)
⊢ (0, abbc)
⊢ (1, bbc)

AUTOMATES FINIS

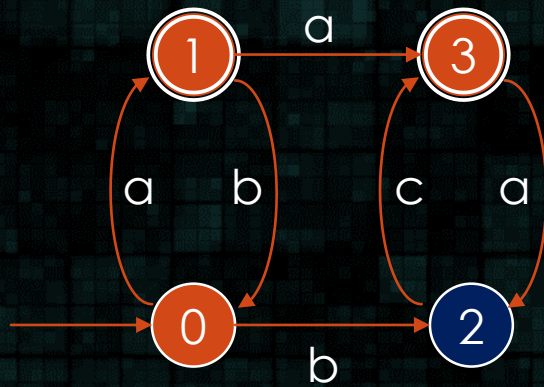
- Reconnaissance :



⊢ (0, ababbc)
⊢ (1, babbbc)
⊢ (0, abbc)
⊢ (1, bbc)
⊢ (0, bc)

AUTOMATES FINIS

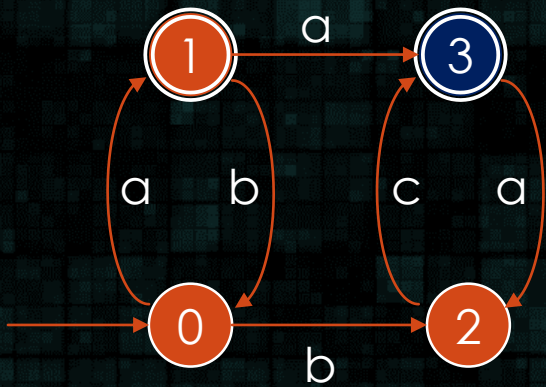
- Reconnaissance :



⊢ (0, ababbc)
⊢ (1, babbc)
⊢ (0, abbc)
⊢ (1, bbc)
⊢ (0, bc)
⊢ (2, c)

AUTOMATES FINIS

- Reconnaissance :



(0, ababbc)
⊢ (1, babbc)
⊢ (0, abbc)
⊢ (1, bbc)
⊢ (0, bc)
⊢ (2, c)
⊢ (3, ε)

AUTOMATES FINIS

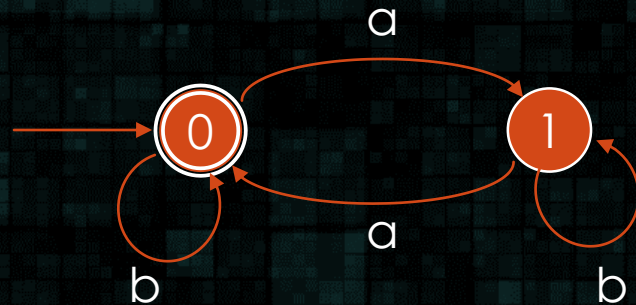
- $L1 = \{ m \in \{ a, b \}^* \}$



- $G1 = \langle \{ S \}, \{ a, b \}, \{ S \rightarrow aS \mid bS \mid \varepsilon \}, S \rangle$

AUTOMATES FINIS

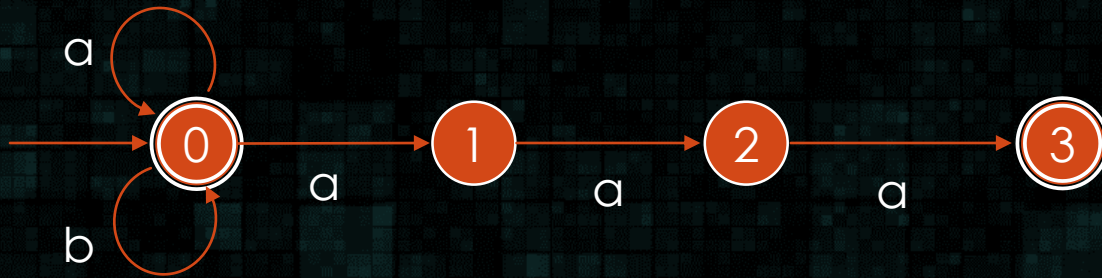
- $L2 = \{ m \in \{ a, b \}^* \mid |m|_a \bmod 2 = 0 \}$



- $G2 = \langle \{ S, T \}, \{ a, b \}, \{ S \rightarrow aT \mid bS \mid \epsilon, T \rightarrow aS \mid bT \}, S \rangle$

AUTOMATES FINIS

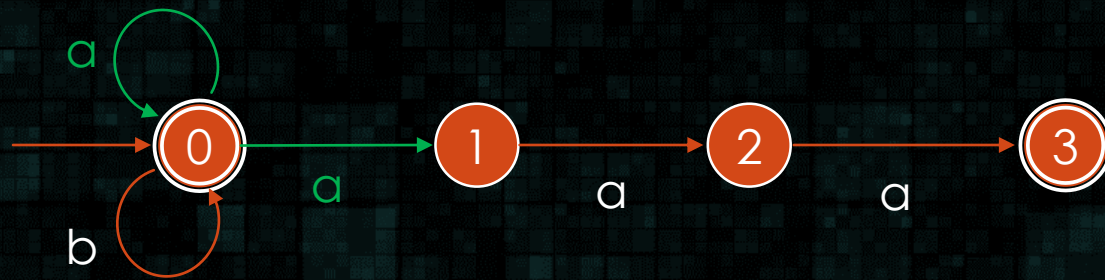
- $L3 = \{ xaaa \mid x \in \{ a,b \}^* \}$



- $G3 = \langle \{ S, T, U \}, \{ a, b \}, \{ S \rightarrow aS \mid bS \mid aT, T \rightarrow aU, U \rightarrow a \}, S \rangle$

AUTOMATES FINIS

- $L3 = \{ xaaa \mid x \in \{ a, b \}^* \}$

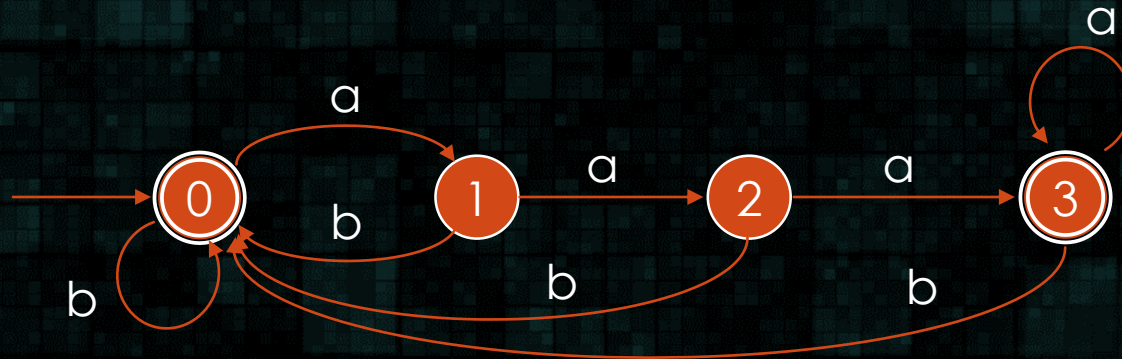


- $G3 = \langle \{ S, T, U \}, \{ a, b \}, \{ S \rightarrow aS \mid bS \mid aT, T \rightarrow aU, U \rightarrow a \}, S \rangle$

Non déterminisme !!!!!

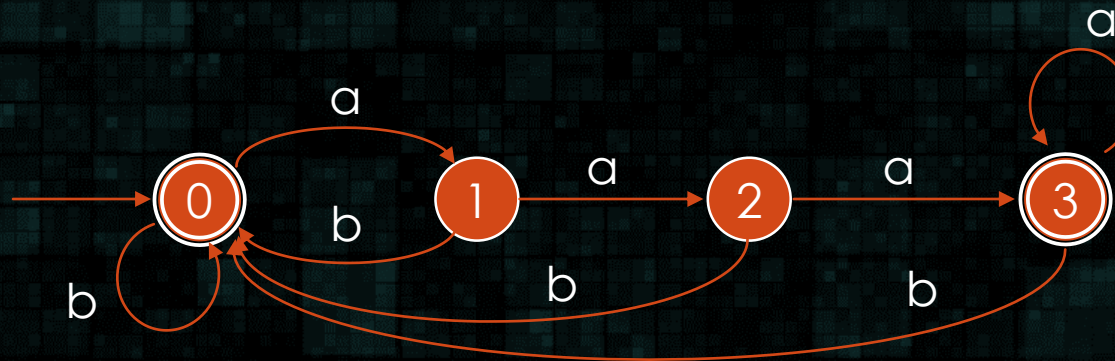
AUTOMATES FINIS

- $L3 = \{ xaaa \mid x \in \{ a,b \}^* \}$



AUTOMATES FINIS

- $L3 = \{ xaaa \mid x \in \{ a,b \}^* \}$



déterministe

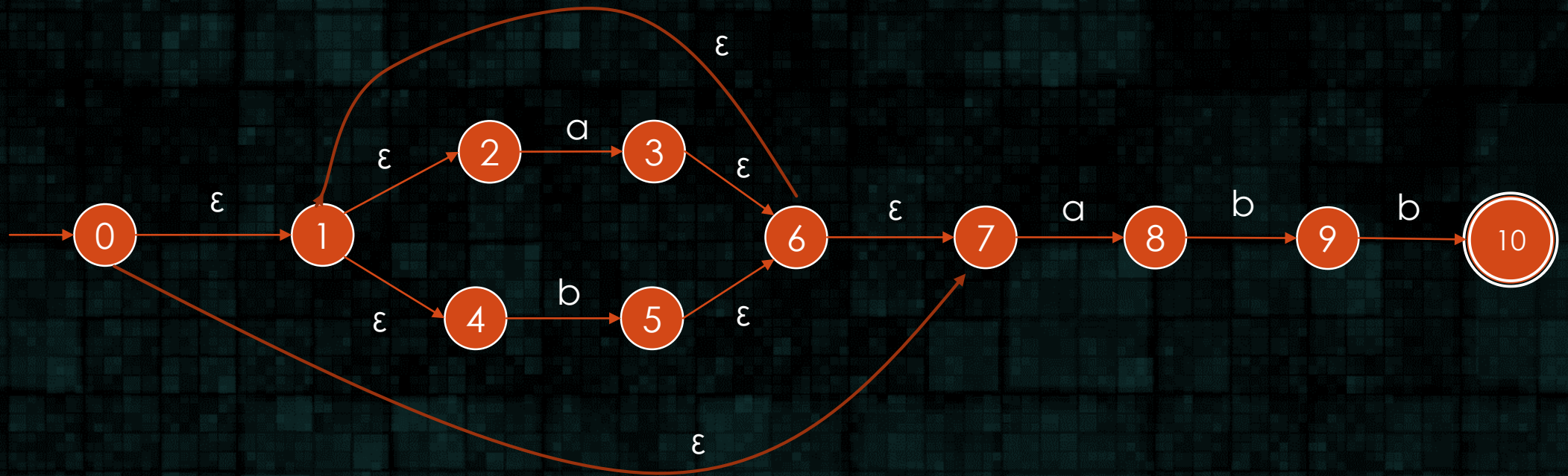
- ⊢ (0, abaabaaa)
- ⊢ (1, baabaaa)
- ⊢ (0, aabaaa)
- ⊢ (1, abaaa)
- ⊢ (2, baaa)
- ⊢ (0, aaa)
- ⊢ (1, aa)
- ⊢ (2, a)
- ⊢ (3, ε)

AUTOMATES FINIS

- Déterminisme
 - Tout langage régulier peut être reconnu par un automate fini déterministe
 - Pour tout automate fini non déterministe A , on peut construire un automate déterministe A' avec $L(A) = L(A')$
 - Prix à payer : dans le pire des cas, $|Q(A')| = 2^{|Q(A)|}$

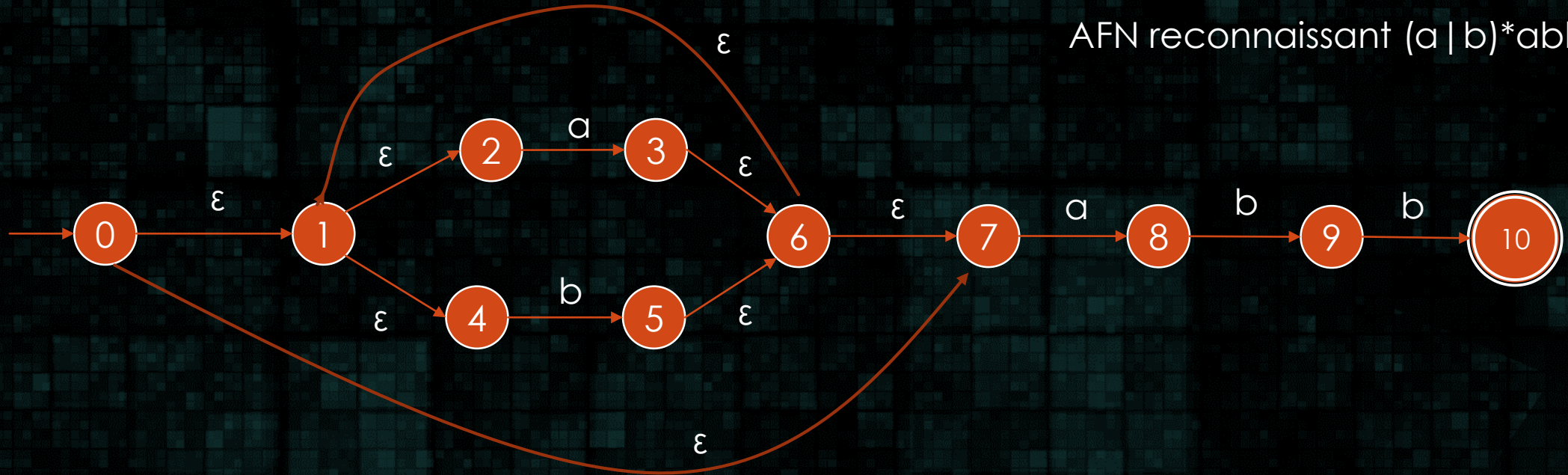
AUTOMATES FINIS

- Détermination d'un AFN en AFD avec ε -transitions



AFN reconnaissant $(a|b)^*abb$

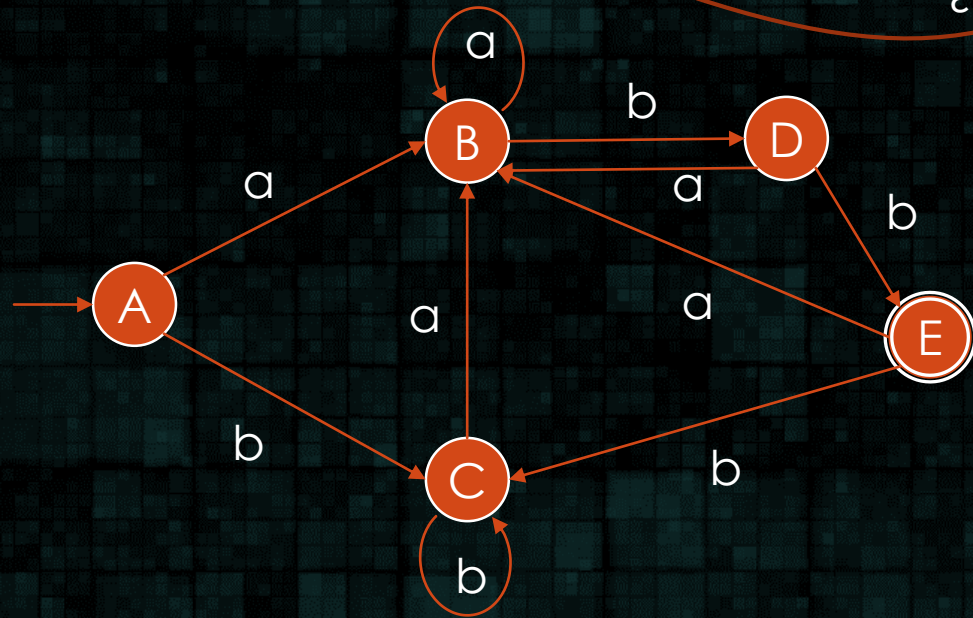
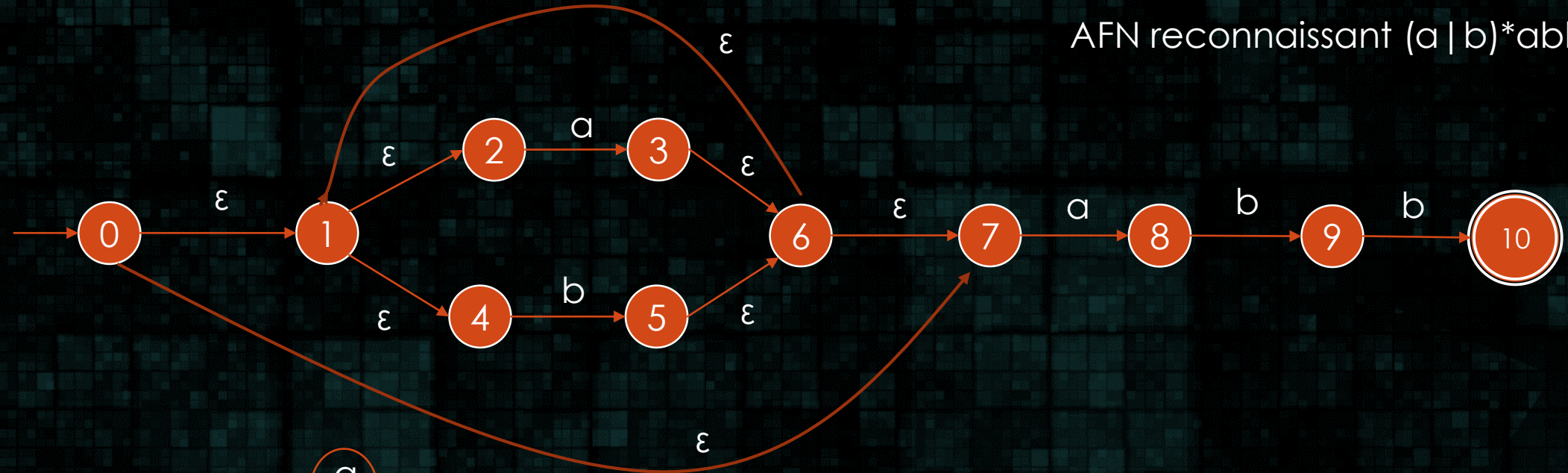
AFN reconnaissant $(a \mid b)^*abb$



Fermeture $\{\{0\}\} = \{0,1,7,2,4\} \rightarrow$ le nouvel état initial est A
 Fermeture $\{\{8,3\}\} = \{8,3,6,1,7,2,4\} \rightarrow$ le nouvel état est B
 Fermeture $\{\{5\}\} = \{5,6,7,1,2,4\} \rightarrow$ le nouvel état est C
 Fermeture $\{\{5,9\}\} = \{5,9,6,7,1,2,4\} \rightarrow$ le nouvel état est D
 Fermeture $\{\{5,10\}\} = \{5,10,6,7,1,2,4\} \rightarrow$ le nouvel état est E

États AFD	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

AFN reconnaissant $(a \mid b)^*abb$



États AFD

→ A
B
C
D
E

	a	b
→ A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
→ 0	0,1	0
1	2	2
2	3	3
← 3	-	-

	a	b
0	0,1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
→ 0	0,1	0
1	2	2
2	3	3
← 3	-	-

	a	b
0	0,1	0
0,1	0,1,2	0,2

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
→ 0	0,1	0
1	2	2
2	3	3
← 3	-	-

	a	b
0	0,1	0
0,1	0,1,2	0,2
0,1,2	0,1,2,3	0,2,3
0,2	0,1,3	0,3

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	0,1	0
0,1	0,1,2	0,2
0,1,2	0,1,2,3	0,2,3
0,2	0,1,3	0,3
0,1,2,3	0,1,2,3	0,2,3
0,2,3	0,1,3	0,3
0,1,3	0,1,2	0,2
0,3	0,1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
→ 0	0,1	0
1	2	2
2	3	3
← 3	-	-



	a	b
0	0,1	0
0,1	0,1,2	0,2
0,1,2	0,1,2,3	0,2,3
0,2	0,1,3	0,3
0,1,2,3	0,1,2,3	0,2,3
0,2,3	0,1,3	0,3
0,1,3	0,1,2	0,2
0,3	0,1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	0,1,2	0,2
0,1,2	0,1,2,3	0,2,3
0,2	0,1,3	0,3
0,1,2,3	0,1,2,3	0,2,3
0,2,3	0,1,3	0,3
0,1,3	0,1,2	0,2
0,3	1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	2	0,2
2	0,1,2,3	0,2,3
0,2	0,1,3	0,3
0,1,2,3	0,1,2,3	0,2,3
0,2,3	0,1,3	0,3
0,1,3	2	0,2
0,3	1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	2	3
2	0,1,2,3	0,2,3
3	0,1,3	0,3
0,1,2,3	0,1,2,3	0,2,3
0,2,3	0,1,3	0,3
0,1,3	2	3
0,3	1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	2	3
2	4	0,2,3
3	0,1,3	0,3
4	4	0,2,3
0,2,3	0,1,3	0,3
0,1,3	2	3
0,3	1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	2	3
2	4	5
3	0,1,3	0,3
4	4	5
5	0,1,3	0,3
0,1,3	2	3
0,3	1	0

AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



	a	b
0	1	0
1	2	3
2	4	5
3	6	0,3
4	4	5
5	6	0,3
6	2	3
0,3	1	0



AUTOMATES FINIS

- La détermination sans les ε -transitions

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-

	a	b
0	1	0
1	2	3
2	4	5
3	6	7
4	4	5
5	6	7
6	2	3
7	1	0

AUTOMATES FINIS

■ Minimalisation d'un AFD →

	a	b	≈ 0	1,2,4,5	3,6
1	2	4			
2	3	6			
← 3	3	3			
4	5	2			
5	3	6			
← 6	6	4			

On construit les classes d'équivalences de $\approx 0, \approx 1, \dots, \approx n$ pas à pas :

1. on commence par ≈ 0 = séparer les états d'acceptation des autres
2. les classes de ≈ 1 sont obtenues en séparant, dans chaque classe de ≈ 0 , les états qui sont envoyés par un même symbole sur des classes de ≈ 0 différentes.
3. On calcule de même ≈ 2 en fonction de ≈ 1 et, de proche en proche, toutes les relations $\approx n$.

AUTOMATES FINIS

■ Minimalisation d'un AFD

	a	b
1	2	4
2	3	6
3	3	3
4	5	2
5	3	6
6	6	4

≈ 0	1,2,4,5		3,6	
≈ 1	1,4	2,5	3	6

On construit les classes d'équivalences de $\approx 0, \approx 1, \dots, \approx n$ pas à pas :

1. on commence par ≈ 0 = séparer les états d'acceptation des autres
2. les classes de ≈ 1 sont obtenues en séparant, dans chaque classe de ≈ 0 , les états qui sont envoyés par un même symbole sur des classes de ≈ 0 différentes.
3. On calcule de même ≈ 2 en fonction de ≈ 1 et, de proche en proche, toutes les relations $\approx n$.

AUTOMATES FINIS

■ Minimalisation d'un AFD

	a	b
1	2	4
2	3	6
3	3	3
4	5	2
5	3	6
6	6	4

≈ 0	1,2,4,5		3,6	
≈ 1	1,4	2,5	3	6
≈ 2	1	4	2,5	3
			6	

On construit les classes d'équivalences de $\approx 0, \approx 1, \dots, \approx n$ pas à pas :

1. on commence par ≈ 0 = séparer les états d'acceptation des autres
2. les classes de ≈ 1 sont obtenues en séparant, dans chaque classe de ≈ 0 , les états qui sont envoyés par un même symbole sur des classes de ≈ 0 différentes.
3. On calcule de même ≈ 2 en fonction de ≈ 1 et, de proche en proche, toutes les relations $\approx n$.

AUTOMATES FINIS

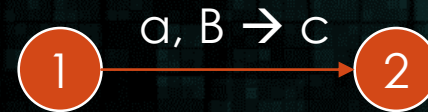
- Limite des automates finis
 - Certains langages ne peuvent pas être reconnus par les automates finis (ne peuvent être engendrés par une grammaire régulière)
 - Exemple : $L = \{ a^n b^n \mid n \geq 0 \}$
 - Il faut mémoriser le nombre de a que l'on a lu pour vérifier que le mot possède autant de b.
 - Pour mémoriser un nombre potentiellement infini de a, il faut un ensemble infini d'états !

AUTOMATES À PILE

- Un automate à pile :
 - Forme simple de mémoire : une pile.
 - Mode de stockage LIFO.
 - On ne peut accéder qu'à l'élément se trouvant au sommet de la pile.
 - Deux opérations possibles :
 - empiler : ajouter un élément au sommet.
 - dépiler : enlever l'élément se trouvant au sommet.
 - Elle commence avec un symbole de fond de pile \perp , que l'on ne peut pas dépiler.
- La pile permet de stocker de l'information sans forcément multiplier le nombre d'états

AUTOMATES À PILE

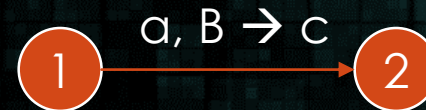
- Représentation graphique



- Si l'automate est en 1, et que la tête de lecture est sur a, l'automate :
 - Décale la tête de lecture d'une case vers la droite
 - Dépile B (B doit être présent au sommet de la pile)
 - Empile c
 - Va en 2

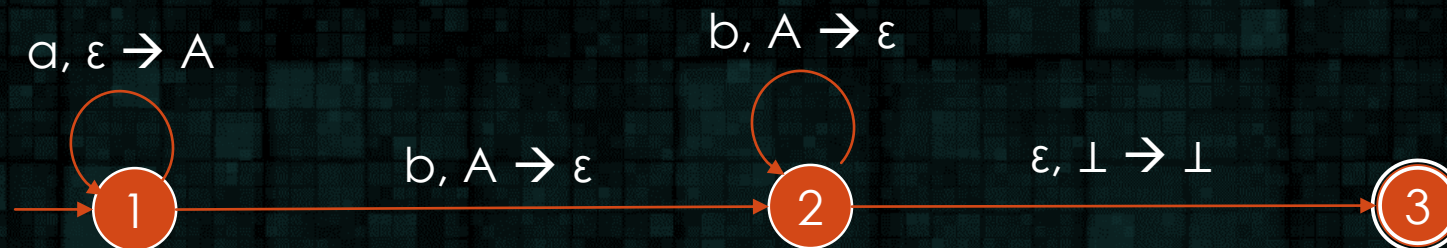
AUTOMATES À PILE

- Représentation graphique



- cas particuliers

- Si $a = \varepsilon$, l'automate peut franchir cet arc sans lire de symbole.
- Si $B = \varepsilon$, l'automate peut franchir cet arc indépendamment du symbole se trouvant en sommet de pile (et ne le dépile pas).
- Si $c = \varepsilon$, l'automate peut franchir cet arc sans rien empiler



AUTOMATES À PILE

- Définition formelle
 - Un automate à pile est un 6-uplet $\langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$
 - Q est l'ensemble des états
 - Σ est l'alphabet d'entrée
 - Γ est l'alphabet de symboles de pile (en particulier, $\perp \in \Gamma$)
 - δ est la fonction de transition
 - $q_0 \in Q$ est l'état initial
 - $F \subseteq Q$ est l'ensemble des états d'acceptation

AUTOMATES À PILE

- Configurations

- $A = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$

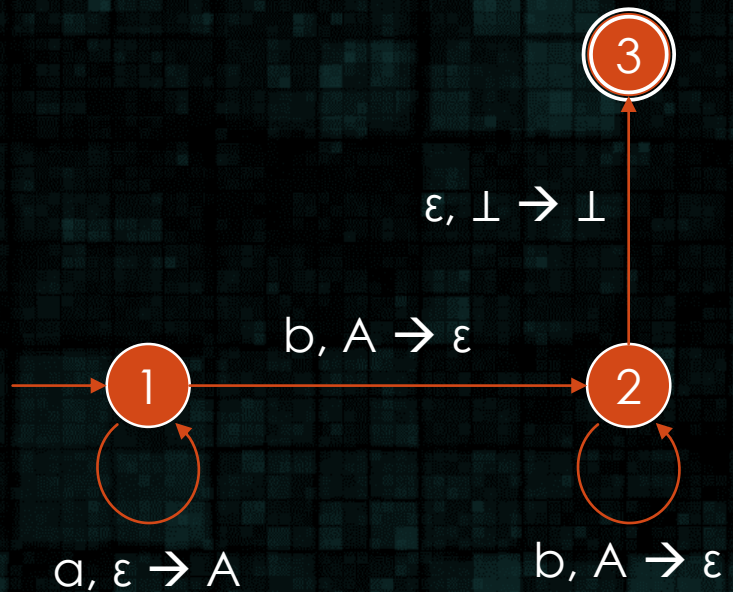
- Configuration :

- q représente l'état courant de l'unité de contrôle
 - m est la partie du mot à reconnaître non encore lue.
 - Le premier symbole de m (le plus à gauche) est celui qui se trouve sous la tête de lecture.
 - Si $m = \epsilon$ alors tout le mot a été lu.
 - α représente le contenu de la pile. Le symbole le plus à gauche est le sommet de la pile. Si $\alpha = \epsilon$ alors la pile est vide.

- Configuration initiale : (q_0, m, \perp) où m est le mot à reconnaître

- Configuration d'acceptation : (q, ϵ, \perp) avec $q \in F$

AUTOMATES À PILE



$\langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$

$Q = \{ 1, 2, 3 \}$

$\Sigma = \{ a, b \}$

$\Gamma = \{ A, \perp \}$

$\delta(1, a, \varepsilon) = \{ (1, A) \}$

$\delta(1, b, A) = \{ (2, \varepsilon) \}$

$\delta(2, b, A) = \{ (2, \varepsilon) \}$

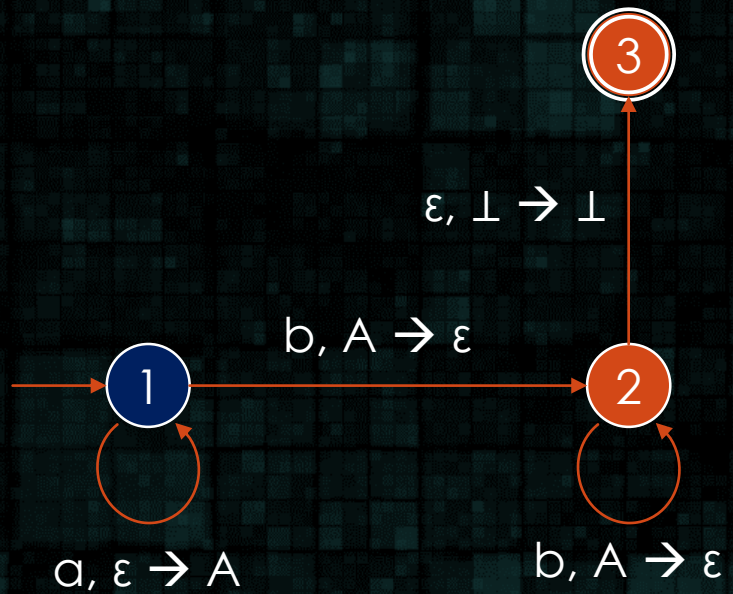
$\delta(2, \varepsilon, \perp) = \{ (3, \perp) \}$

$q_0 = 1$

$F = \{ 3 \}$

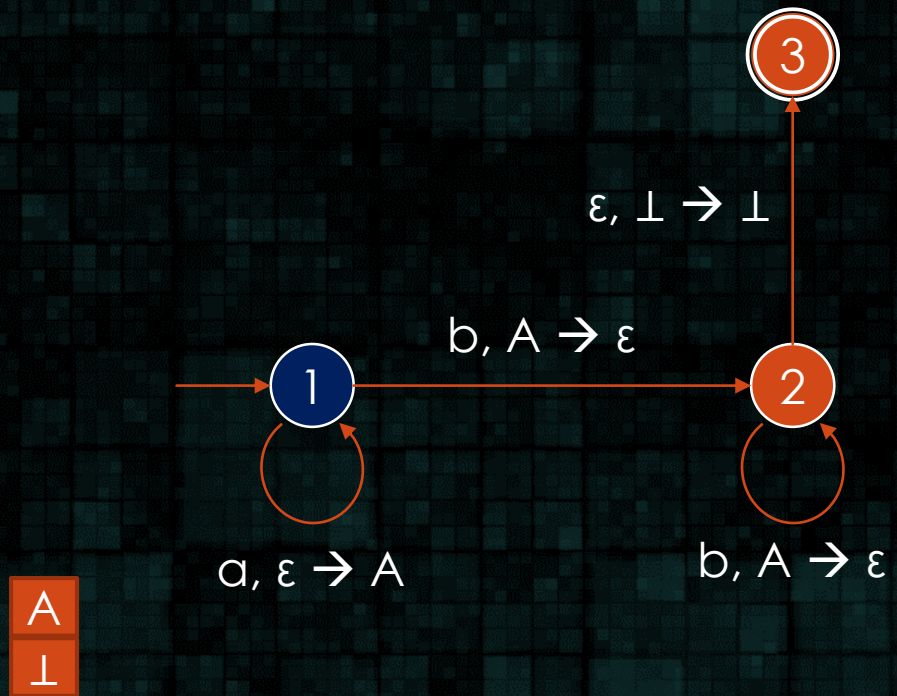
AUTOMATES À PILE

$(1, aaabbb, \perp)$



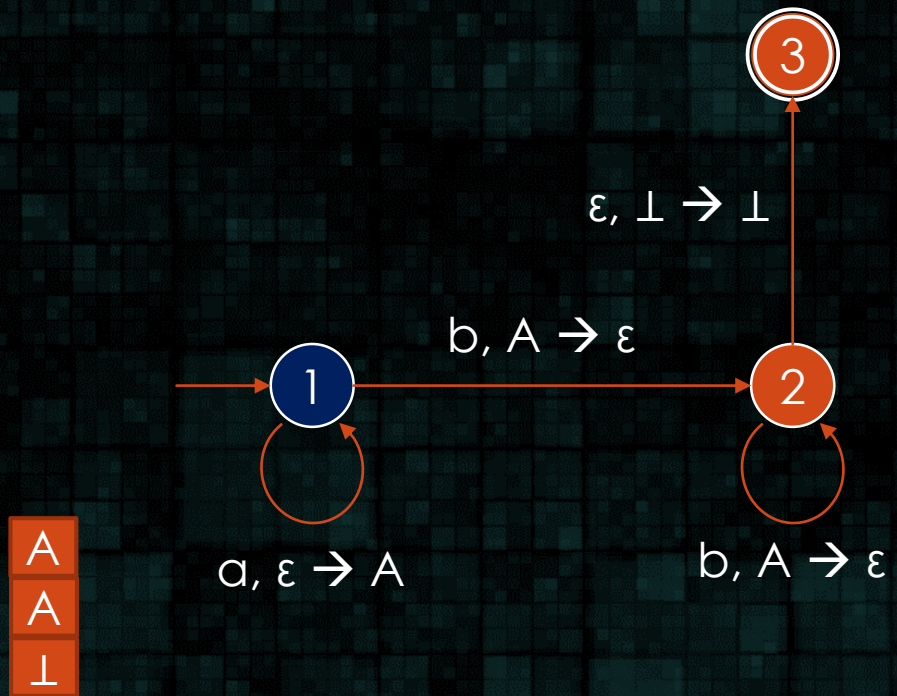
\perp

AUTOMATES À PILE



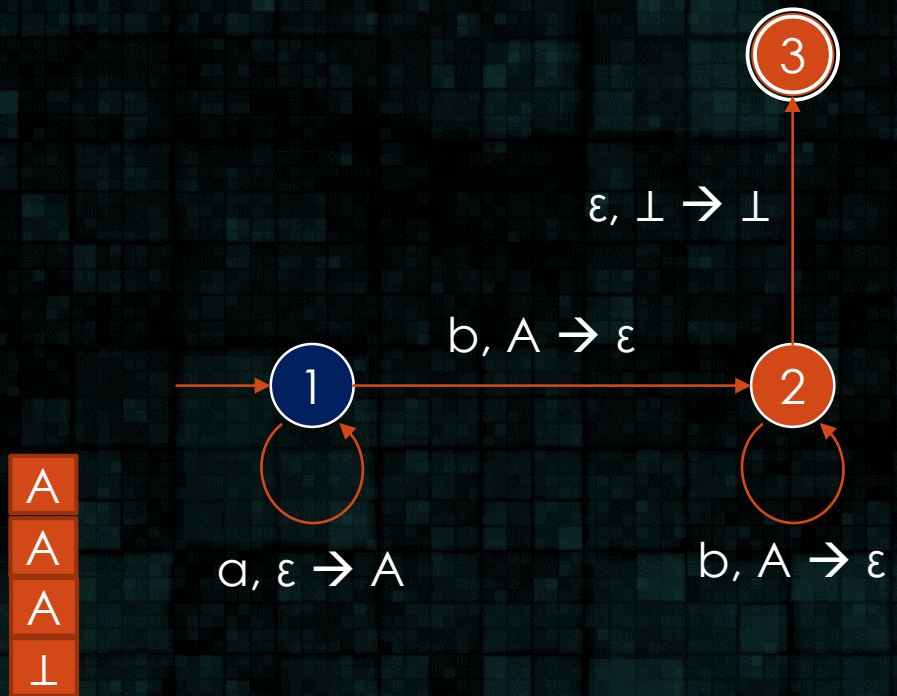
$\vdash (1, aaabbbb, \perp)$
 $\vdash (1, aaabbb, A\perp)$

AUTOMATES À PILE



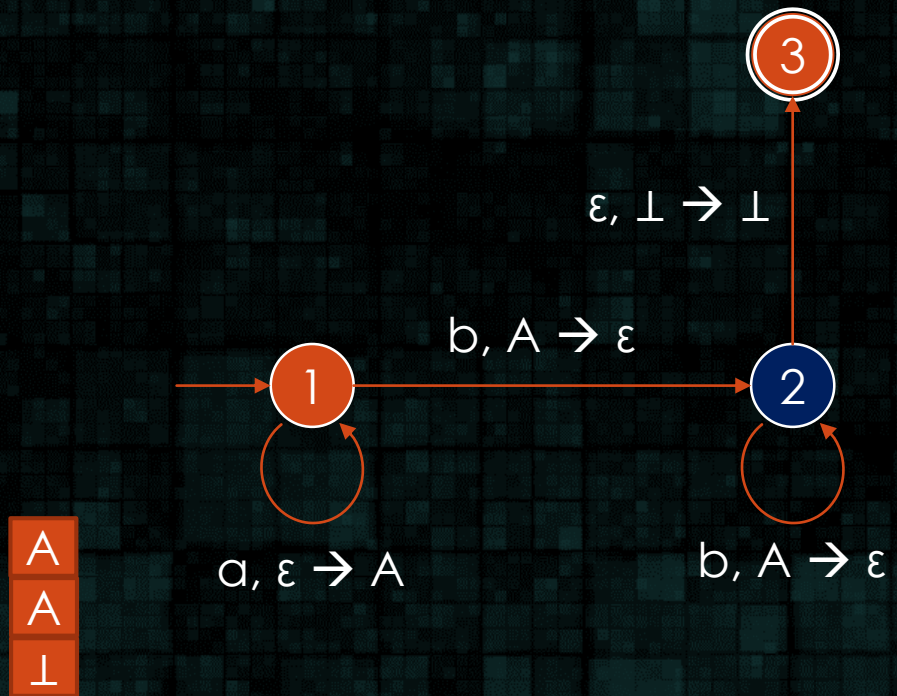
- $\vdash (1, aaabbbb, \perp)$
- $\vdash (1, aabbbb, A\perp)$
- $\vdash (1, abbbb, AA\perp)$

AUTOMATES À PILE



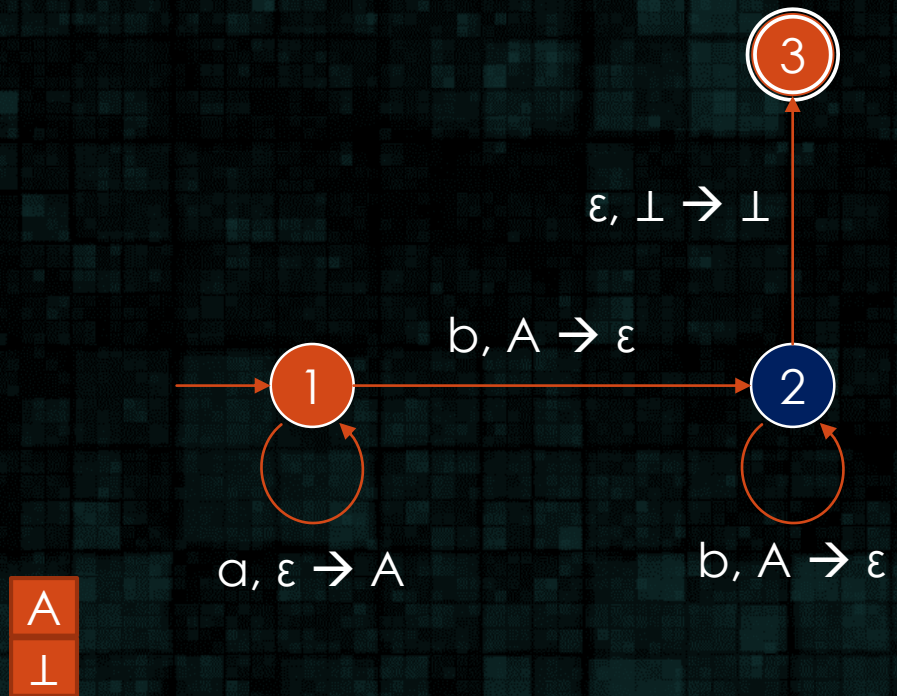
- $\vdash (1, aaabbbb, \perp)$
- $\vdash (1, aabbbb, A\perp)$
- $\vdash (1, abbbb, AA\perp)$
- $\vdash (1, bbbb, AAA\perp)$

AUTOMATES À PILE



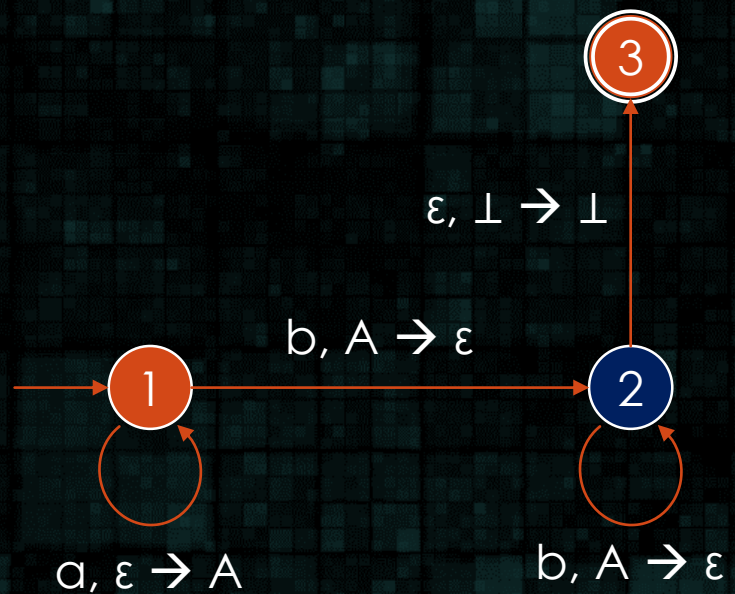
- $\vdash (1, aaabbbb, \perp)$
- $\vdash (1, aabbbb, A\perp)$
- $\vdash (1, abbbb, AA\perp)$
- $\vdash (1, bbbb, AAA\perp)$
- $\vdash (2, bb, AA\perp)$

AUTOMATES À PILE



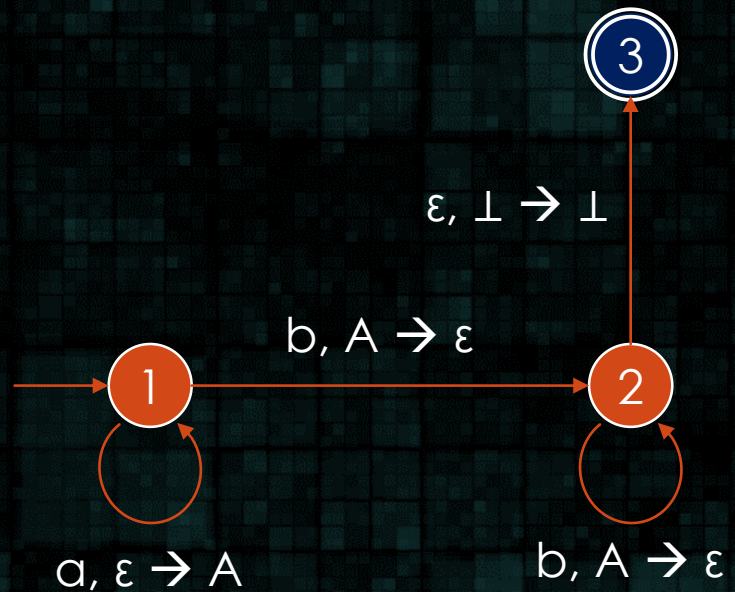
- $\vdash (1, aaabbbb, \perp)$
- $\vdash (1, aabbbb, A\perp)$
- $\vdash (1, abbbb, AA\perp)$
- $\vdash (1, bbbb, AAA\perp)$
- $\vdash (2, bb, AA\perp)$
- $\vdash (2, b, A\perp)$

AUTOMATES À PILE



- ⊢ (1, aaabbbb, ⊥)
- ⊢ (1, aabbbb, A⊥)
- ⊢ (1, abbbb, AA⊥)
- ⊢ (1, bbbb, AAA⊥)
- ⊢ (2, bb, AA⊥)
- ⊢ (2, b, A⊥)
- ⊢ (2, ε, ⊥)

AUTOMATES À PILE



⊢ (1, aaabbbb, ⊥)
⊢ (1, aabbbb, A⊥)
⊢ (1, abbbb, AA⊥)
⊢ (1, bbbb, AAA⊥)
⊢ (2, bb, AA⊥)
⊢ (2, b, A⊥)
⊢ (2, ε, ⊥)
⊢ (3, ε, ⊥)

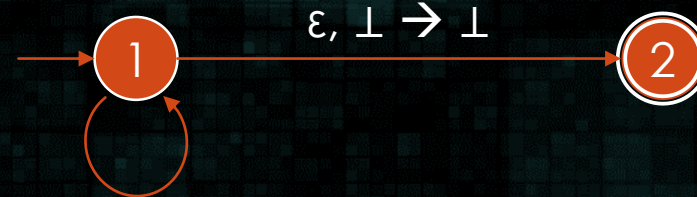
AUTOMATES À PILE

- $L = \{ m \in \{a, b\}^*, |m|_a = |m|_b \}$

$b, B \rightarrow BB$

$b, \perp \rightarrow B\perp$

$b, A \rightarrow \varepsilon$



$a, B \rightarrow \varepsilon$

$a, \perp \rightarrow A\perp$

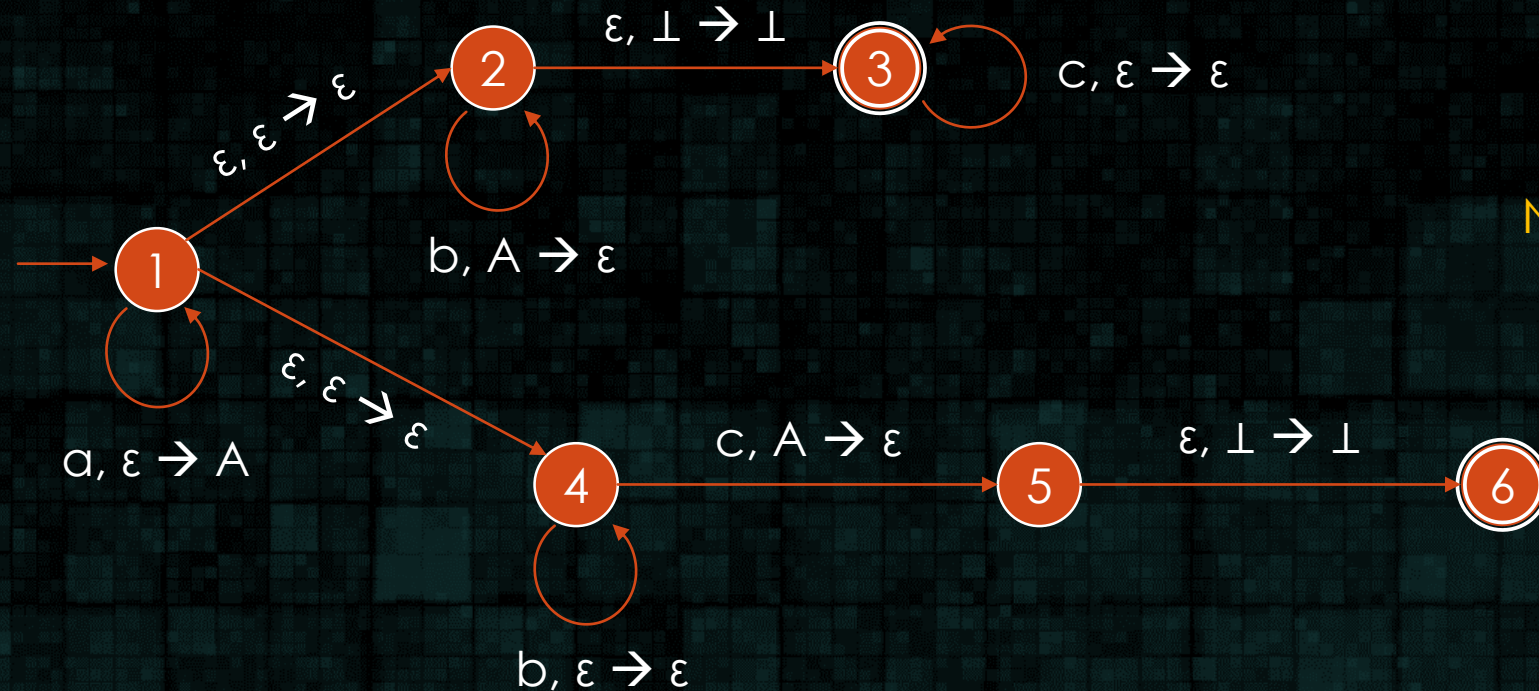
$a, A \rightarrow AA$

- $G = \langle \{S\}, \{a, b\}, \{S \rightarrow aSb \mid bSa \mid SS \mid \varepsilon\}, S \rangle$

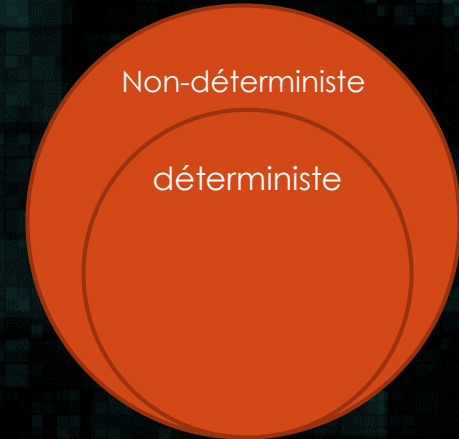
AUTOMATES À PILE

- $L = \{ a^i b^j c^k \mid i \geq 0, i = j \text{ ou } i = k \}$

$b, B \rightarrow BB$
 $b, \perp \rightarrow B\perp$
 $b, A \rightarrow \varepsilon$



Non déterministe !!



AUTOMATES À PILE


- Limites des automates à pile
 - Certains langages ne peuvent être reconnus par les automates à pile (ne peuvent être engendrés par une grammaire hors-contexte).
 - Exemple : le langage $m\#m$ avec $m \in \{0, 1\}^*$:
 - Un automate lit le premier m et le stocke dans la pile.
 - Il lit le premier symbole du second m .
 - Comment vérifier qu'il est identique au symbole se trouvant au fond de la pile ?

MACHINE DE TURING

- Proches des automates finis, mais avec une mémoire infinie et à accès direct.
- Modèle plus proche d'un ordinateur.
- Une machine de Turing (MT) peut faire tout ce qu'un ordinateur peut faire.
- Thèse de Church-Turing : tout traitement réalisable par un algorithme peut être accompli par une machine de Turing.

MACHINE DE TURING

- Caractéristiques

- Elle possède une tête de lecture/écriture pouvant se déplacer vers la gauche et vers la droite.
- Au départ, la bande contient le mot à reconnaître et possède des  dans toutes les autres cases
- Une MT peut lire et écrire sur la bande de lecture/écriture.
- La tête de lecture/écriture peut se déplacer vers la droite et vers la gauche.
- La bande de lecture écriture est infinie.
- Lorsque la MT atteint l'état d'acceptation ou l'état de rejet, elle s'arrête et accepte ou rejette le mot.
- Si la MT n'atteint pas l'état d'acceptation ou de rejet, elle peut continuer indéfiniment.

MACHINE DE TURING

- Représentation graphique



- La machine est en 1, la tête de lecture est sur a, elle :
 - écrit un b sur la bande (à la place du a),
 - décale la tête de lecture d'une case vers la droite (D)
 - va en 2.
- Cas particuliers :
 - $a \rightarrow G$: la machine n'écrit rien.
 - a : la machine n'écrit rien et ne bouge plus (elle arrive généralement dans un état final).

MACHINE DE TURING

- Définition

- Une MT est un octuplet $\langle Q, \Sigma, \Gamma, \square, \delta, q_0, q_A, q_R \rangle$ où :
 - Q est l'ensemble des états,
 - Σ est l'alphabet de l'entrée (qui ne contient pas le symbole spécial \square),
 - Γ est l'alphabet de la bande ($\square \in \Gamma$ et $\Sigma \subseteq \Gamma$),
 - δ est la fonction de transition :
 - $q_0 \in Q$ est l'état initial,
 - $q_A \in Q$ est l'état d'acceptation,
 - $q_R \in Q$ est l'état de rejet, avec $q_R \neq q_A$.

MACHINE DE TURING

- Exemple

- Principe d'une machine reconnaissant $L = \{ m\#m \mid m \in \{0, 1\}^* \}$.
 - Fait des allers-retours entre les deux occurrences de m pour vérifier qu'elles contiennent bien le même symbole. Si ce n'est pas le cas, ou qu'un $\#$ supplémentaire est détecté, va dans l'état de rejet. Les symboles sont éliminés au fur et à mesure qu'ils sont vérifiés (par le symbole x à gauche, et z à droite).
 - Lorsque tous les symboles à gauche de $\#$ ont été éliminés, vérifie qu'il ne reste plus de symboles à droite de $\#$. Si c'est le cas, va dans l'état d'acceptation, sinon va dans l'état de rejet.

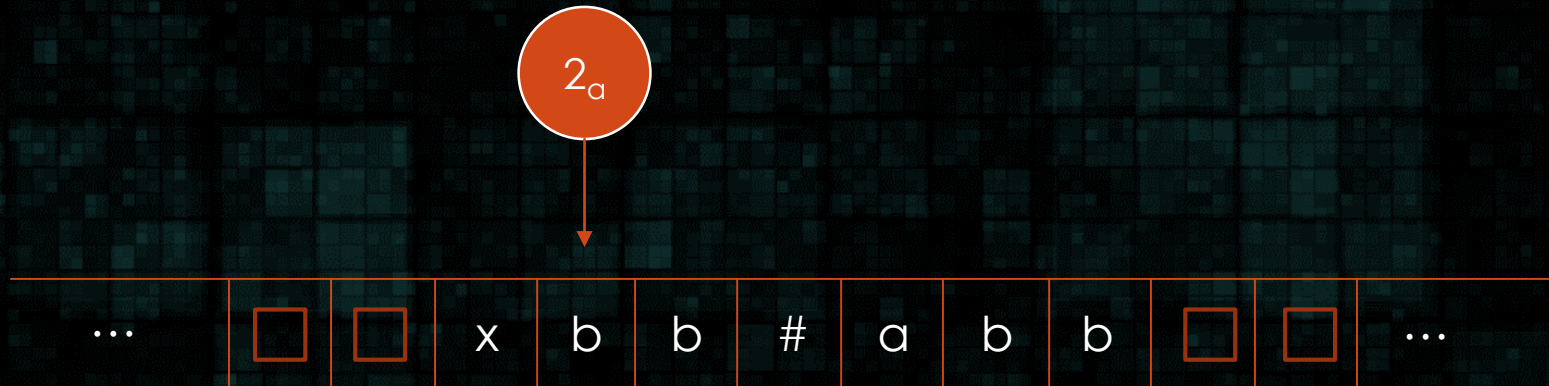
MACHINE DE TURING



$$\delta(1,a) = (x, 2_a, D)$$

enregistre a

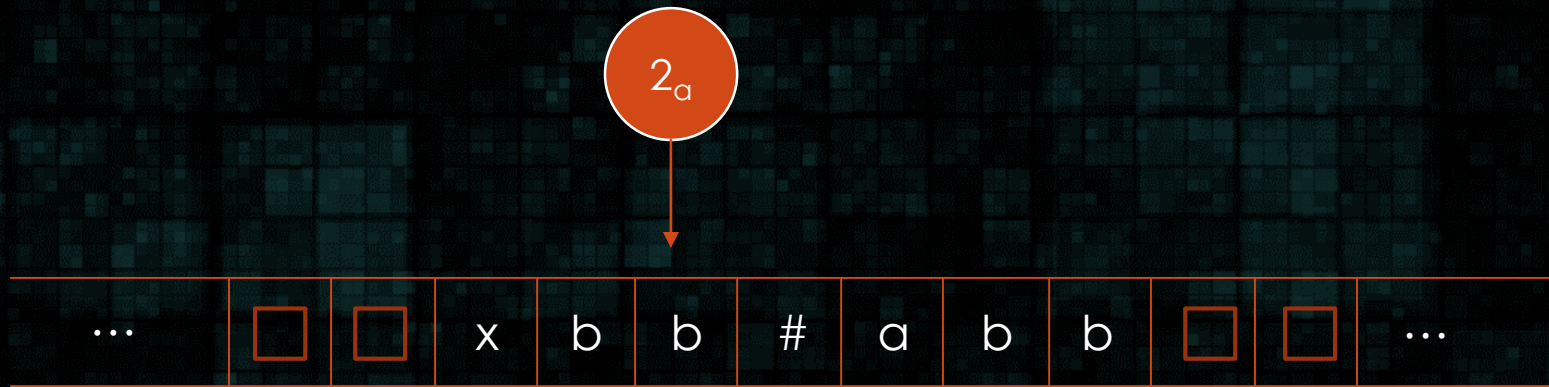
MACHINE DE TURING



$$\begin{aligned}\delta(1, a) &= (x, 2_a, D) \\ \delta(2_a, a) &= (a, 2_a, D) \text{ pour } a \neq \#\end{aligned}$$

enregistre a
cherche #

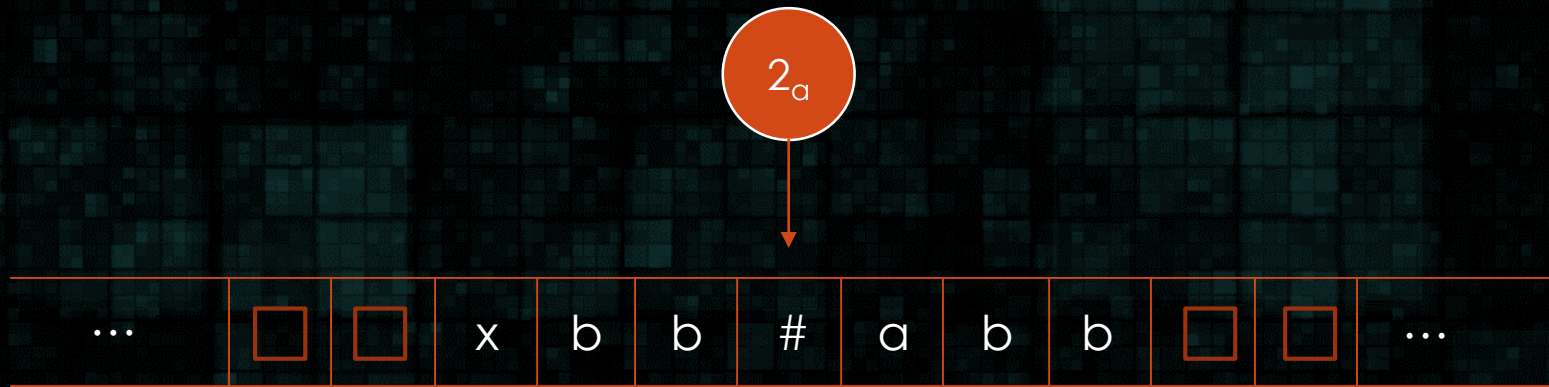
MACHINE DE TURING



$$\begin{aligned}\delta(1, a) &= (x, 2_a, D) \\ \delta(2_a, a) &= (a, 2_a, D) \text{ pour } a \neq \#\end{aligned}$$

enregistre a
cherche #

MACHINE DE TURING



$$\delta(1, a) = (x, 2_q, D)$$

$$\delta(2_q, a) = (a, 2_q, D) \text{ pour } a \neq \#$$

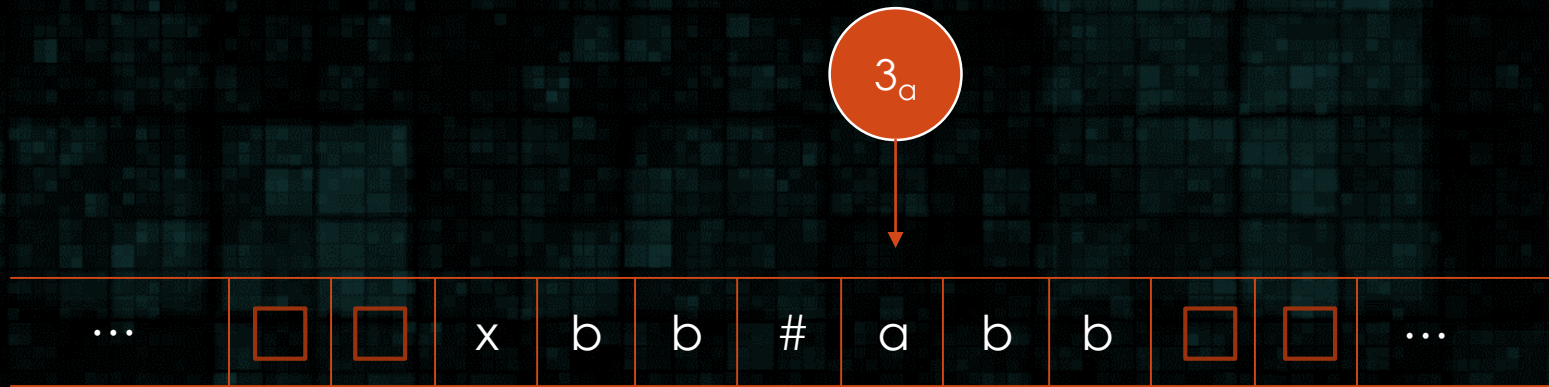
$$\delta(2_q, \#) = (\#, 3_q, D)$$

enregistre a

cherche #

3_q cherche une lettre non-z

MACHINE DE TURING



$\delta(1, a) = (x, 2_a, D)$

$\delta(2_a, a) = (a, 2_a, D)$ pour $a \neq \#$

$\delta(2_a, \#) = (\#, 3_a, D)$

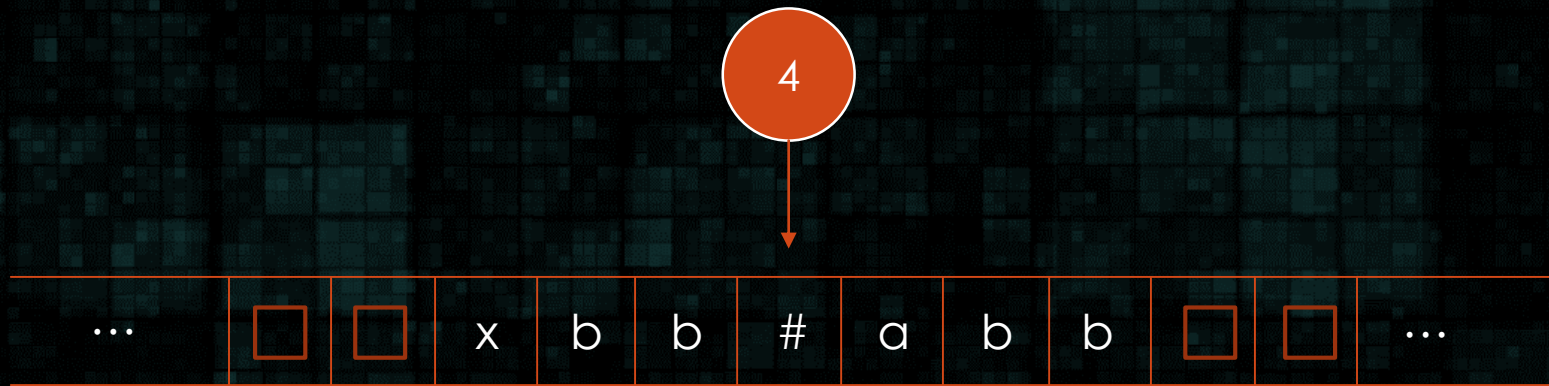
$\delta(3_a, a) = (z, 4, G)$

enregistre a

cherche #

3_a cherche une lettre non-z
vérifie la lettre

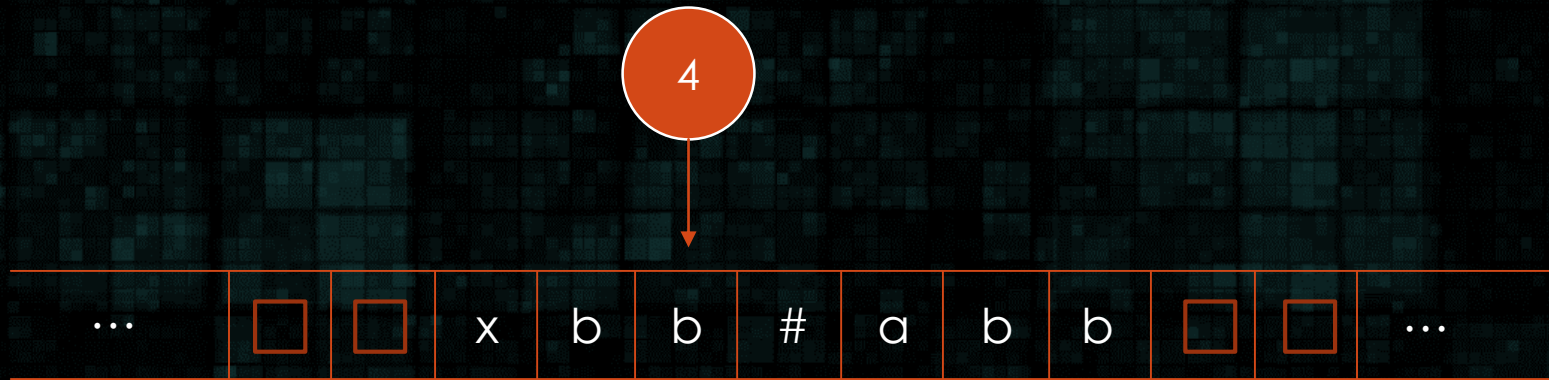
MACHINE DE TURING



$\delta(1, a) = (x, 2_a, D)$
 $\delta(2_a, a) = (a, 2_a, D)$ pour $a \neq \#$
 $\delta(2_a, \#) = (\#, 3_a, D)$
 $\delta(3_a, a) = (z, 4, G)$
 $\delta(4, a) = (a, 4, G)$ pour $a \neq x$

enregistre a
cherche #
 3_a cherche une lettre non-z
vérifie la lettre
revient vers x

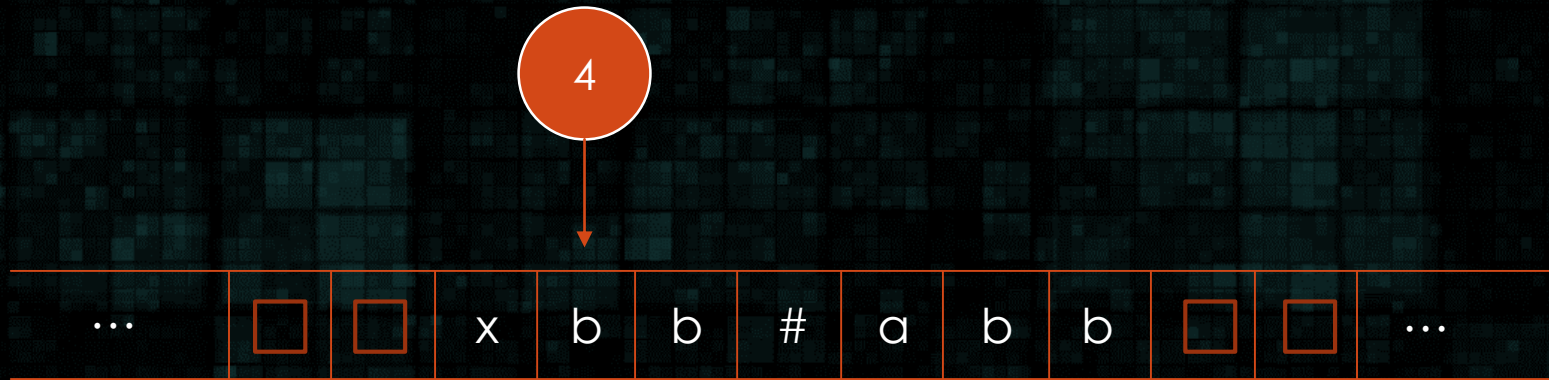
MACHINE DE TURING



$\delta(1, a) = (x, 2_a, D)$
 $\delta(2_a, a) = (a, 2_a, D)$ pour $a \neq \#$
 $\delta(2_a, \#) = (\#, 3_a, D)$
 $\delta(3_a, a) = (z, 4, G)$
 $\delta(4, a) = (a, 4, G)$ pour $a \neq x$

enregistre a
cherche #
 3_a cherche une lettre non-z
vérifie la lettre
revient vers x

MACHINE DE TURING



$\delta(1, a) = (x, 2_a, D)$
 $\delta(2_a, a) = (a, 2_a, D)$ pour $a \neq \#$
 $\delta(2_a, \#) = (\#, 3_a, D)$
 $\delta(3_a, a) = (z, 4, G)$
 $\delta(4, a) = (a, 4, G)$ pour $a \neq x$

enregistre a
cherche #
 3_a cherche une lettre non-z
vérifie la lettre
revient vers x

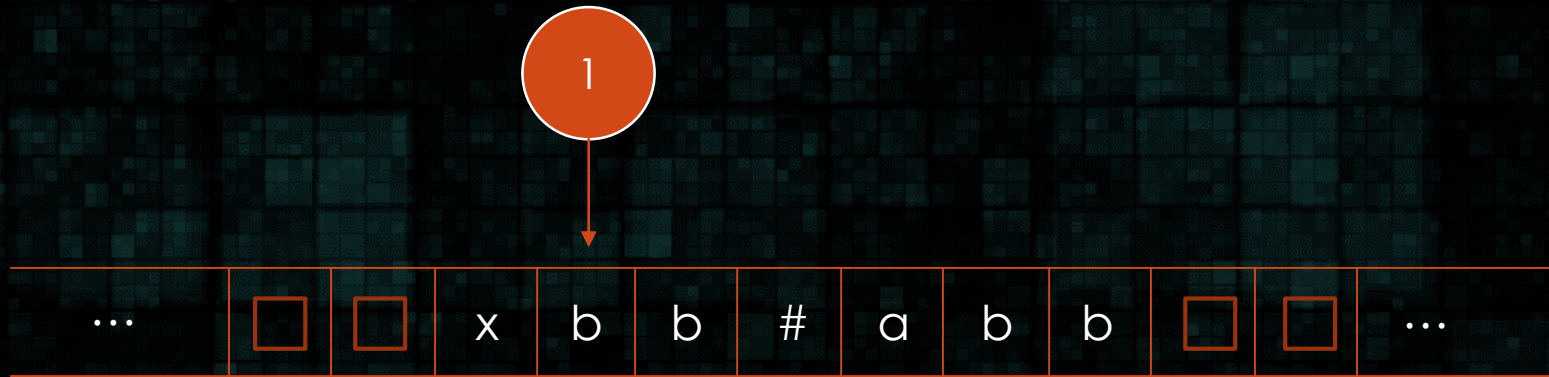
MACHINE DE TURING



$\delta(1, a) = (x, 2_q, D)$
 $\delta(2_q, a) = (a, 2_q, D)$ pour $a \neq \#$
 $\delta(2_q, \#) = (\#, 3_q, D)$
 $\delta(3_q, a) = (z, 4, G)$
 $\delta(4, a) = (a, 4, G)$ pour $a \neq x$
 $\delta(4, x) = (x, 1, D)$

enregistre a
cherche #
 3_q cherche une lettre non-z
vérifie la lettre
revient vers x
relance le processus

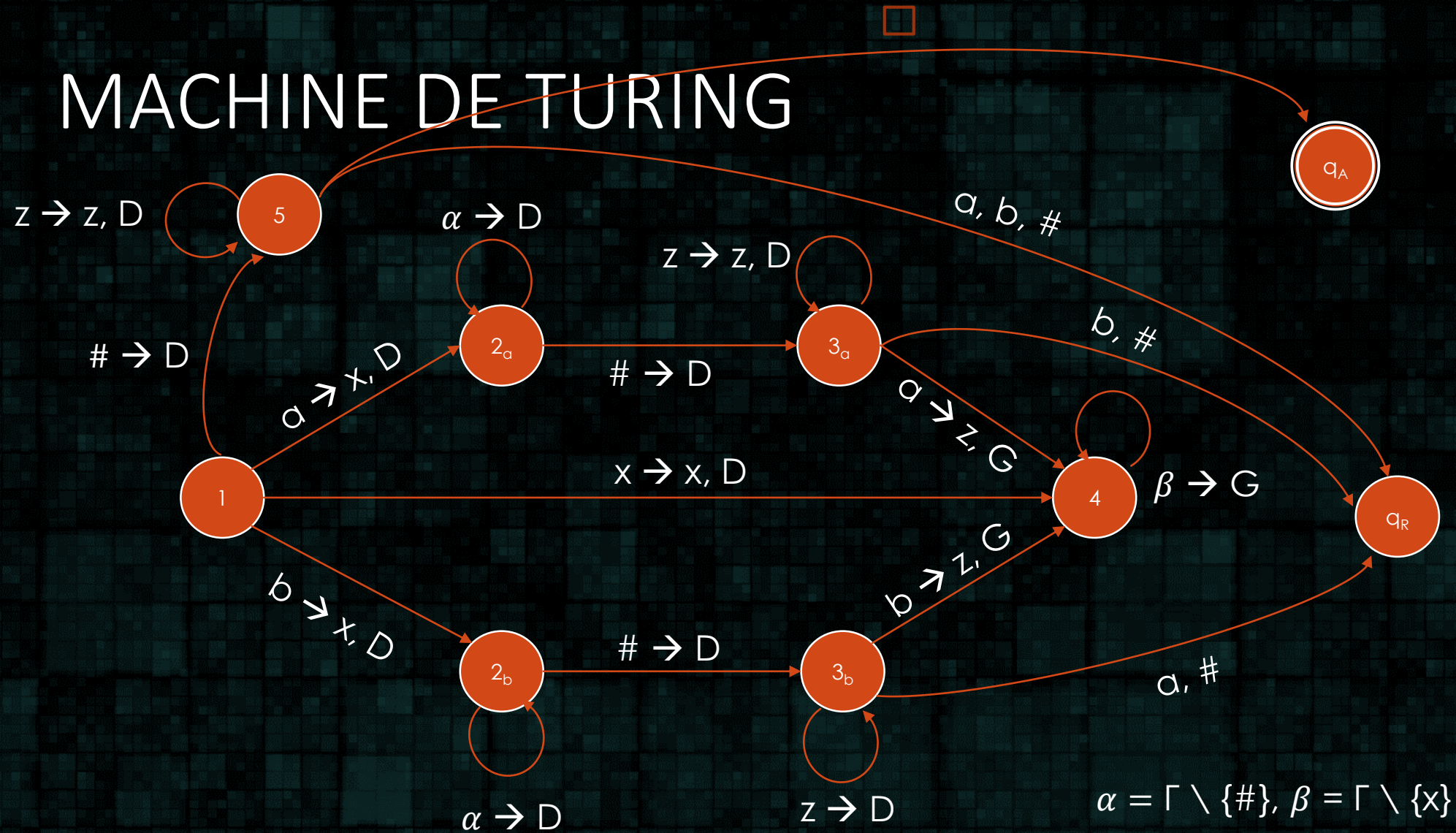
MACHINE DE TURING



$\delta(1, a) = (x, 2_a, D)$
 $\delta(2_a, a) = (a, 2_a, D)$ pour $a \neq \#$
 $\delta(2_a, \#) = (\#, 3_a, D)$
 $\delta(3_a, a) = (z, 4, G)$
 $\delta(4, a) = (a, 4, G)$ pour $a \neq x$
 $\delta(4, x) = (x, 1, D)$
 $\delta(1, b) = (x, 2_b, D) \dots$

enregistre a
cherche #
 3_a cherche une lettre non-z
vérifie la lettre
revient vers x
relance le processus

MACHINE DE TURING



MACHINE DE TURING

- Déterminisme
 - Pour toute MTA non déterministe, il existe une MTA' telle que $L(A) = L(A')$.
 - Le non déterminisme n'augmente pas la puissance du modèle des MT.

MACHINE DE TURING

- Langages récursivement énumérables
 - Un langage est récursivement énumérable si et seulement si il existe une MT qui le reconnaît.
 - Un langage est récursivement énumérable si et seulement si il existe une MT déterministe qui le reconnaît.