CHAPITRE

AUTOMATES ET CALCULABILITÉ DES LANGAGES

4

Le troisième moyen permettant de répondre à la question d'appartenance d'un mot w à un langage L : Ce sont les automates. Un automate est une machine qui, après avoir accompli une série d'opérations sur une séquence de caractères (un mot), a la capacité de fournir une réponse sous forme de "oui" (accepté) ou "non" (rejeté) à cette question.

4.1 Qu'est ce qu'un automate ?

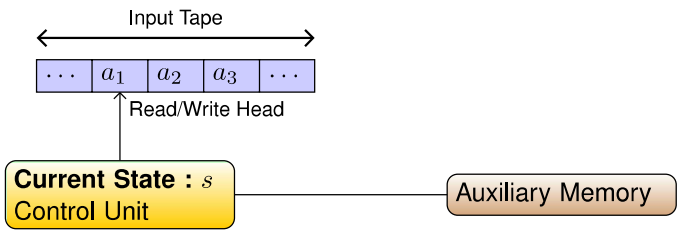
Définition 4.1.1 Automate

Un automate est une machine abstraite qui permet de lire (analyser) un mot w et de répondre à la question : "w appartient-il à un langage ?". La machine de Turing représente l'abstraction universelle de la notion d'automate programmable.

L'objectif final de la construction des automates est la construction d'un programme équivalent qui analyse des mots. Cependant, nous verrons que la faisabilité (**décidabilité**) de cette analyse et le temps (**complexité**) ne sont pas toujours garanties

4.1.1 Composants

FIGURE 4.1 – Composants d'un automate



Un automate est généralement composé de :

— Une bande ou un ruban en entrée finie ou infinie (selon le type d'automate) des deux côtés, sur lequel est inscrit le mot à lire. Le ruban est composé de cases, chacune stocke un seul symbole.

— Une tête de lecture/écriture L/E (selon le type de d'automate) : qui peut lire ou écrire dans une seule case du ruban à la fois. Elle peut également se déplacer à droite ou à gauche sur le ruban (tout dépend du type de d'automate).

— Un organe de commande (unité de contrôle) qui contrôle les actions de la tête de L/E à travers un ensemble fini de transitions (des pas d'exécution)

— Une éventuelle mémoire auxiliaire de stockage (selon le type de l'automate)

4.1.2 Formalisation

Formellement, un automate A contient au minimum :

— Un alphabet X, sur lequel sont définis les mots en entrée.

— Un ensemble non vide d'états Q.

— Un état initial $s_0 \in Q$

— Un ensemble non vide d'états finaux $F \subseteq Q$.

— Une fonction de transition δ qui permet de passer d'une configuration à une autre, permettant de : changer ou non d'état, effectuer une action sur le ruban (éventuellement sur la mémoire).

Grace à cette fonction de transition l'automate peut analyser un mot en passant par une séquence de configurations, partant d'une configuration initiale vers une configuration finale.

Définition 4.1.2 Configuration

Une configuration d'un automate est définie par le triplet (s, a, mem) qui représente les valeurs que peuvent prendre ses différents composants :

- s : C'est létat dans lequel se trouve l'automate
- a : Le symbole couramment lu sur lequel se positionne la tête de L/E
- mem : représente éventuellement le contenu de la mémoire auxiliaire (s'il y en a une).

Définition 4.1.3 Configuration initiale

Une configuration initiale signifie que l'automate se trouve dans l'état initial s_0 et sa tête de L/E positionnée sur le premier symbole du mot à analyser.

Définition 4.1.4 Configuration finale

Une configuration finale signifie que l'automate se trouve dans l'un des états finaux $s_f \in F$ et le mot en question a été entièrement lu

➤ Le passage d'une configuration à une autre est défini par la fonction de transition δ selon l'état en cours de l'automate, le symbole en cours de lecture par la tête de L/E et éventuellement le contenu de la mémoire auxiliaire.

Déduction 4.1.1 $w \in L(A)$ d'un point de vue automate

Un mot $w \in L(A)$ si'il est accepté par un automate A.

- w est accepté par A si, à partir d'une configuration initiale, A arrive à une configuration finale en passant par une succession de configurations intermédiaires qui lui permet de lire entièrement w.
- L'ensemble de tous les mots acceptés par A représente le langage $L(A)$. Autrement dit A accepte uniquement les mots de L.

4.2 Classification des automates

Comme les grammaires et selon la hiérarchie de Chomsky, les automates peuvent être classés en 4 classes. Chaque classe accepte un type de langages. Le tableau suivant résume les classes de grammaires, les langages générés et les types d'automates qui les acceptent.

TABLE 4.1 – Les types de grammaires, langages et automates

Grammaire	Langage	Automate
Type 3 (régulière)	Régulier ou rationnel	Automate à états finis
Type 2 (hors-contexte)	Algébrique	Automate à pile
Type 1 (contextuelle)	Contextuel	Automate à borne linéaire
Type 0	Récursivement énumérable	Machine de Turing

Automate à états finis (AEF)

C'est le type d'automates qui acceptent les langages réguliers, générés par les grammaires de type 3. un AEF est caractérisé par :

— Une bande (ruban) en entrée qui est finie ;

— Une tête en lecture seule et dans un seul sens de gauche à droite. Donc pas d'écriture sur la bande

— Pas de mémoire auxiliaire.

Automate à pile (AP)

C'est le type d'automates qui acceptent les langages algébriques générés par les grammaires de type 2. La structure d'un AP est similaire à celle d'un AEF sauf qu'il dispose en plus d'une mémoire organisée sous forme d'une seule pile infinie.

Automate à borne linèaire (ABL)

C'est le type d'automates qui acceptent les langages contextuels générés par les grammaires de type 1. un ABL est caractérisé par :

— Une bande en entrée finie accessible en lecture/écriture.

— Une lecture dans les deux sens.

— Pas de mémoire auxiliaire.

Machine de Turing (MT)

Une MT représente le type d'automates universels qui acceptent les langages générés par les grammaires de type 0, ou plutôt les langages que l'on sait reconnaître (ou énumérer) avec une machine de Turing (langages récursivement énumérable). La structure d'une MT est la même que celle d'un ABL mais la bande en entrée est infinie.

4.3 Décidabilité et Complexité

Les langages que génèrent les grammaires de type 0 ne se ressemblent pas tous. Certains sont dits plus complexes que d'autres. En réalité, la construction des machines de Turing acceptant ces langages n'est tout à fait faisable et leur comportement n'est pas le mêmes pour tous les langages.

Remarque 4.3.1

- Les machines de Turing sont utilisées comme modèle de calcul formel pour les notions de semi-calculabilité et calculabilité.
- De l'autre coté, la théorie des fonctions récursives a été élaborée par Gödel et Kleene comme

modèle de fonction calculable basé sur un ensemble de fonctions et primitives de base : fonctions constantes, composition, récursion, etc.

C'est la raison pour laquelle nous étudions brièvement la notion de décidabilité (calculabilité) d'un langage : Langage décidable, semi-décidable, en utilisant la notion d'algorithme qui simule l'existence ou non d'une machine de Turing pour un langage.
▶ Donc certains langages ne possèdent même pas de machines de Turing qui les acceptent.

Mais il faut noter aussi que le terme algorithme reste vague et peut être considéré comme "n'importe quel programme écrit dans un langage de programmation" (Python, Java, etc.) qui s'exécuterait dans un environnement sans limitation matérielle (taille mémoire, espace de stockage). On dit qu'un algorithme A se termine pour une entrée x si l'exécution de cet algorithme s'arrête après un nombre fini d'étapes. Cela va nous conduire à étudier aussi la complexité d'analyse d'un mot (d'un langage).
▶ Certaines machine de Turing nécessitent un temps d'analyse très important.

4.3.1 Décidabilité (Calculabilité) des langages

Définition 4.3.1 Langage récursivement énumérable

Un langage L est défini sur un alphabet X, est qualifié comme récursivement énumérable (semi-décidable ou semi-calculable) s'il existe un algorithme A qui énumère les mots de L. Autrement dit, il exsite une machine de Turing qui accepte tout mot de L.

▶ C'est-à-dire :

— $\forall w \in L$, l'algorithme A est capable de lire entièrement w et de s'arrêter dans un état final en moyennant un nombre fini de configurations, en produisant une réponse positive (Vraie). Dans le cas contraire s'il existe certains mots $w \in L$ dont l'analyse ne peut jamais aboutir (l'algorithme tourne à l'infini), alors le langage qui n'est pas récursivement énumérable.

— Si $w \notin L$, alors l'algorithme A soit il se termine et produit la sortie Faux, ou bien A ne se termine jamais.

Théorème 4.3.1

Soit un alphabet X. Il existe un langage $L \subset X^*$ qui n'est pas semi-décidable.

Exemple

Prenons l'exemple de l'ensemble des programmes en langages C qui se terminent. un des résultats fondamentaux de la théorie de la calculabilité c'est le "problème d'arrêt", qui pose la question suivante : "Peut-on construire une machine de Turing qui, en un temps fini, peut décider si un programme (mot) écrit en langage C s'arrêtera (c'est-à-dire se terminera) pour une entrée donnée ?"
La réponse est non et cela découle du théorème d'indécidabilité de l'arrêt, énoncé par Alan Turing. Un théorème qui démontre que, en général, il n'existe pas de procédure algorithmique (une méthode calculable) qui puisse résoudre ce problème pour tous les programmes C et toutes les entrées possibles.

Théorème 4.3.2 Turing

Le langage de l'arrêt est semi-décidable mais non décidable.

▶ Donc, on peut comprendre que la définition d'un langage récursivement énumérable ne garantit pas que l'algorithme d'analyse d'un mot qui ne lui appartient puisse toujours se terminer. Cette limitation conduit à une définition plus rigoureuse d'un sous-type de langages : décidables (récursifs).

Définition 4.3.2 Langage récursif

Un langage L défini sur un alphabet X est dit récursif (décidable) s'il existe algorithme A qui, en prenant un mot $w \in X^*$ en entrée, se termine et produit une réponse soit Vrai si $w \in L$ ou Faux sinon. On dit alors que l'algorithme A décide le langage L. Autrement dit : si on arrive à trouver une machine de Turing qui peut décider de l'appartenance ou non à L de n'importe quel mot $w \in X^*$ en moyennant un nombre fini de configurations.

C'est-à-dire que la procédure de l'analyse est décidable et se termine toujours quel que soit son résultat final et quelle que soit l'entrée.

Remarque 4.3.2

- Les langages non semi-décidables ont peu d'intérêt en pratique car il n'existe pas d'algorithme d'énumération pour ces langages .
- Un langages semi-décidable mais non décidable est beaucoup plus intéressant car on peut énumérer l'ensemble de ses mots, mais on ne peut pas en fournir d'algorithme de reconnaissance

4.3.2 Complexité des langages

Selon les conclusions obtenues dans la section précédente, on s'intéressera uniquement aux langages L récursifs (décidables). pour étudier la complexité des langages. Une complexité qui sera étudiée à travers un automate qui se traduit par un algorithme (programme) de reconnaissance équivalent qui analyse une entrée (un mot) et reconnaît les mots de L.

Définition 4.3.3 Complexité

La **complexité** d'analyse d'un mot d'une certaine taille n (par ce programme qui s'exécute sur une machine) se donne par le nombre de ressources (temps et espace) mobilisées lors de cette analyse :

— **Complexité en temps :** C'est le temps nécessaire pour analyser un mot.

— **Complexité en espace :** C'est la taille du ruban et la mémoire auxiliaire nécessaires pour analyser le mot

Critère de complexité d'un langage

Selon la définition des langages récursifs, leur analyse nécessite un nombre de configurations fini $f(n)$ et cela veut dire :

- Le temps d'analyse des mots d'un langage récursif est fini. Pour le quantifier, on suppose que le passage d'une configuration à une autre se fait en un temps fixe t. Alors le temps d'analyse sera $t.(f(n)-1)$.
- La taille de la mémoire ainsi que la taille du ruban utilisés pour analyser un mot d'un langage récursif sont finies. Pour les quantifier, il s'agit de calculer le nombre de cases utilisées sur le ruban ainsi que la taille de la mémoire utilisée en fonction de $f(n)$

Déduction 4.3.1

Donc le nombre de configurations possibles pour analyser un mot $w \in L$ représente un critère essentiel pour étudier la complexité de L.

Mais un nombre de configurations fini ne signifie pas que les langages récursifs consomment peu de ressources. Bien que l'analyse d'un mot $w \in L$ se termine en un temps fini, cela n'empêche pas cette analyse de prendre un temps considérable, pouvant s'étendre sur des milliards de siècles, même si l'algorithme d'analyse s'execute sur le plus rapide des ordinateurs.

Déduction 4.3.2

- La complexité des langages récursifs peut aller de la plus simple, avec une seule configuration : $f(n) = 1$, à une fonction exponentielle $(f(n) = t^n)$ ce qui rend la machine (l'algorithme) tout à fait inexploitable.
- Si on peut construire plusieurs automates (machines de Turing) acceptant le même langage L, plusieurs complexités seront associées à L. Mais seulement, la machine la moins complexe, la plus optimale (celle ayant le plus faible $f(n)$ en moyenne) représente la complexité de L.

Remarque 4.3.3 Complexité selon les classes des langages

Selon la classification de Chomsky :

— Les langages réguliers ont une complexité dite linéaire, donnée par $n \times t$, $(f(n) = n)$

— Seulement un sous-ensemble des langages algébriques qui possèdent une complexité linéaire. C'est la sous-classe des langages algébriques linéaires à laquelle appartiennent les langages de programmation usuels.

— La complexité des langages contextuels et celle des langages récursifs (qui ne sont pas des langages réguliers, ni algébriques) est beaucoup plus difficile à quantifier.