Université Abdelmalek Essaadi École Nationale des Sciences Appliquées - Tétouan

Projet Théorie des Langages Partie 1 :

Lecture et affichage d'un automate

Travail réalisé par l'équipe :
Bouarguan Abdellah
Ben Yacoub Nizar
El Ghazouani Marouane
Cherradi Ilyass
El Younoussi Nafisa

Période de réalisation : 23 Février 2026 - 29 Février 2026

Table des matières

1 Introduction

Dans le cadre du module de Théorie des Langages et Compilation, l'objectif global de ce projet est de développer un programme permettant de remplir une partie de la table des symboles à partir soit d'une expression régulière, soit d'un automate fini décrit sous forme d'un fichier dot. Ce rapport détaille la "Partie 1 : Lecture et affichage d'un automate", travail réalisé entre le 23 Février et le 29 Février 2026. Il retrace notre démarche chronologique, de la recherche théorique à l'implémentation finale des structures, en passant par la mise en place d'un environnement de travail collaboratif robuste.

2 Recherche Préliminaire, Outils et Étude de Projets Existants

2.1 Concepts Théoriques

D'un point de vue formel, et comme défini dans le support de cours de Théorie des Langages, un automate fini (ou machine à états finis) est un modèle mathématique de calcul. Il est précisément caractérisé par un 5-uplet (Q, A, δ, q_0, F) où :

- Q représente l'ensemble des états (qui est fini dans le cas d'un automate fini) ;
- A désigne l'alphabet de l'entrée ;
- δ est la fonction de transition, définie formellement comme $\delta : Q \times A \to Q$;
- $q_0 \in Q$ est l'état initial, sachant qu'un automate peut éventuellement posséder plusieurs états initiaux ($q_0 \subseteq Q$) ;
- $F \subseteq Q$ constitue l'ensemble des états d'acceptation (ou états finaux).

En informatique, ces modèles sont fondamentaux dans la conception de compilateurs, l'analyse lexicale, et la vérification de systèmes.

2.2 Format DOT et Écosystème GraphviZ

Pour décrire la structure de nos automates, nous avons opté pour le format `.dot`, conformément aux spécifications du projet. Il s'agit d'un langage de description de graphes en texte brut issu de l'écosystème open-source **GraphviZ**. Ce format intuitif modélise naturellement les états sous forme de nœuds et les transitions sous forme d'arêtes orientées (*edges*).

2.3 Manipulation Visuelle

Afin de vérifier la validité de notre parsing, il était nécessaire de visualiser graphiquement les automates. Sur environnement Linux, nous avons utilisé l'utilitaire de commande `dot` fourni par GraphviZ pour générer des images (PNG ou PDF) à partir de nos fichiers textes. La commande employée était typiquement : `dot -Tpng automate.dot -o automate.png`

Pour faciliter le flux de travail des membres de l'équipe opérant sous environnement Windows, des convertisseurs GraphviZ en ligne, tels que *GraphvizOnline*, ont été utilisés comme alternative efficace sans nécessiter d'installation locale complexe.

2.4 Inspiration et Étude de Projets Existants

Avant d'entamer la phase d'implémentation, une recherche a été menée sur des plateformes de partage de code telles que **GitHub**. L'analyse de projets open-source similaires (notamment des parsers de graphes développés en C et des manipulateurs d'expressions régulières) nous a permis d'identifier les meilleures pratiques de l'industrie. Ces recherches nous ont particulièrement inspirés sur la gestion dynamique de la mémoire et les design patterns d'analyse syntaxique pour le traitement de fichiers texte de manière robuste.

3 Environnement de Travail et Choix Techniques

3.1 Le Langage C

Le langage **C** a été retenu pour le développement de ce projet. Ce choix se justifie d'une part par son adéquation naturelle avec les concepts bas niveau de la théorie des langages (développement d'analyseurs lexicaux et syntaxiques). D'autre part, le C offre un contrôle fin sur la gestion de la mémoire via les pointeurs, une caractéristique indispensable pour optimiser la manipulation des tables de symboles et des graphes volumineux.

3.2 Git et Collaboration sur GitHub

Étant une équipe de 5 membres, l'utilisation d'un système de contrôle de version distribué tel que **Git** (hébergé sur GitHub) était impérative. L'intégralité du code source de notre projet ainsi que l'historique de nos contributions sont accessibles sur notre dépôt public : `https://github.com/AbdellahBouarguan/projet_theorie_langages`.

La collaboration s'est orchestrée autour de la stratégie du *Feature Branching* : chaque développeur travaillait sur une branche isolée pour sa tâche spécifique avant de soumettre une *Pull Request*. La protection de la branche principale (`main`) a été instaurée pour empêcher l'écrasement involontaire du code et s'assurer que seules des implémentations testées y soient fusionnées.

3.3 Standardisation de la Compilation avec CMake

Initialement, la compilation du projet reposait sur un simple `Makefile`. Cependant, une problématique de portabilité s'est rapidement imposée au sein de notre équipe hétérogène (utilisation de Linux , couplée à des systèmes d'exploitation Windows). Les commandes de shell Unix appelées dans le fichier `Make` n'étaient pas reconnues nativement sous Windows.

Pour résoudre ce goulet d'étranglement, nous avons migré vers **CMake**. Cet outil génère un processus de compilation agnostique vis-à-vis du système d'exploitation, créant des `Makefiles` sous Linux et des solutions Visual Studio ou MinGW sous Windows, uniformisant ainsi notre chaîne de build multiplateforme.

4 Organisation et Répartition du Travail

La réussite de cette première phase repose sur une répartition stricte et coordonnée des responsabilités entre les membres de l'équipe :

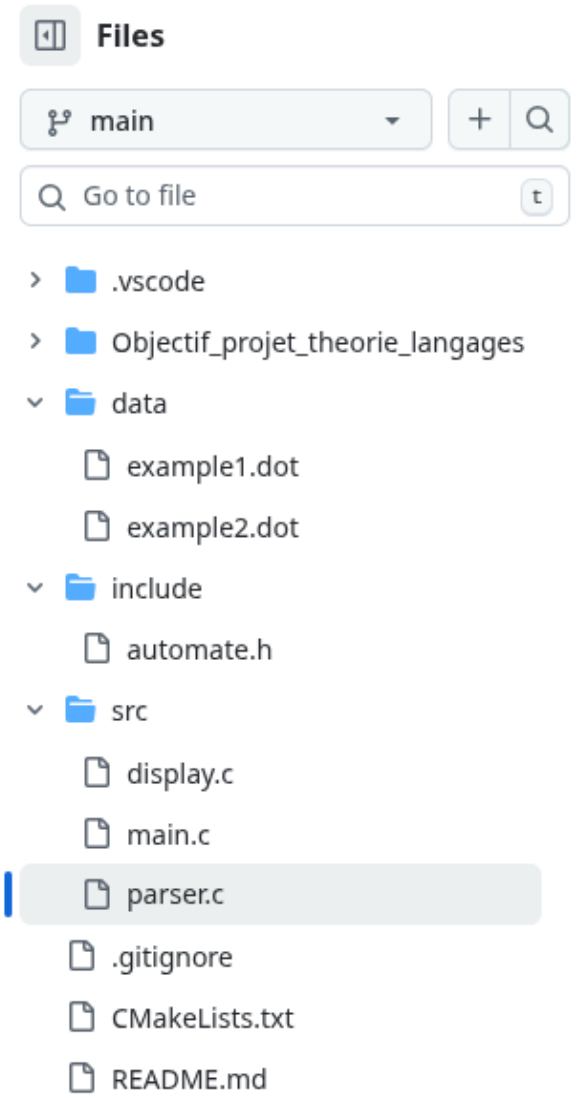

FIGURE 1 – Structure de notre dépôt public sur GitHub

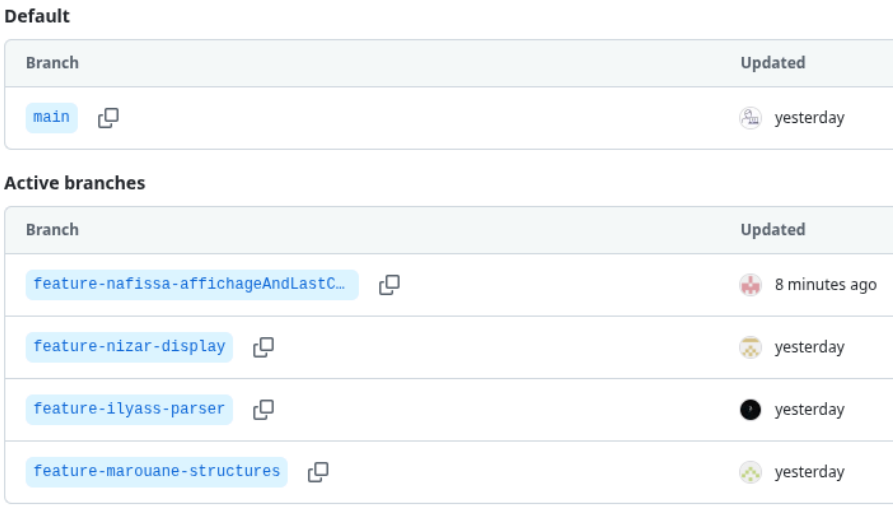

FIGURE 2 – Illustration de notre stratégie de Feature Branching

— **Abdellah** : Chargé de l'initialisation de l'architecture des répertoires et de la configuration de l'infrastructure de base (Git et CMake). Il supervise le bon fonctionnement du projet et la coordination de l'équipe, gère la chaîne de compilation et l'exécution du programme, et assure la rédaction documentaire garantissant la traçabilité des choix techniques.
— **Nizar** : Responsable de la traduction des automates finis vus en cours vers le format DOT à l'aide de l'outil en ligne **GraphvizOnline** (génération des fichiers `data/example1.dot` et `data/example2.dot`). Il a également pris en charge la modélisation mathématique en structures de données logicielles (conception du module `automate.h`) et a contribué au développement du moteur d'affichage (`display.c`).
— **Marouane** : En charge du développement du moteur d'analyse syntaxique permettant la lecture et l'extraction de données depuis les fichiers DOT (`parser.c`). Il a également contribué à la conception de la structure de données (`automate.h`) et a initialisé l'intégration de cet automate au sein du *parser*.
— **Ilyas** : Développeur du module d'affichage console, s'assurant que la machinerie interne soit exposée lisiblement à l'écran (`display.c`). Il a aussi activement participé au développement du module d'analyse syntaxique (`parser.c`).
— **Nafissa** : Responsable de l'intégration globale des modules et conceptrice de l'interface utilisateur interactive (le menu principal dans `main.c`).

5 Conception et Implémentation de la Phase 1

5.1 Structures de données (`automate.h`)

Afin de répondre au premier objectif du cahier des charges demandant de proposer des structures de données permettant l'implémentation d'un automate fini, nous avons défini des structures en C capables de stocker les états, les transitions, l'alphabet, ainsi que la nature de chaque état (initial, final).

La structure `Transition` modélise un arc entre deux états, étiqueté par un symbole de l'alphabet :

```c
// Structure pour une transition
typedef struct {
    int from_etat;
    int to_etat;
    char label[MAX_LABEL_LEN];
} Transition;
```

Listing 1 – Structure `Transition` (extraite de automate.h)

La structure `Automaton` regroupe toutes les informations pertinentes pour définir formellement l'automate (états, transitions, alphabet, booléens pour la nature des états) :

```c
#define MAX_ETATS 100
#define MAX_TRANSITIONS 500
#define MAX_ALPHABET 50
#define MAX_LABEL_LEN 64
// Structure pour l'automate
typedef struct {
    int num_etats;
    int etats[MAX_ETATS];          // Liste des identifiants (ex: 0, 1, 2)
    bool is_final[MAX_ETATS];       // Indique si l'état est final
```

```
    bool is_initial[MAX_ETATS];     // Indique si l'état est initial

    int num_transitions;
    Transition transitions[MAX_TRANSITIONS];

    int num_alphabet;
    char alphabet[MAX_ALPHABET][MAX_LABEL_LEN];
} Automaton;
```

Listing 2 – Structure `Automaton` (extraite de automate.h)

5.2 Lecture et stockage (Parsing dans `parser.c`)

Le deuxième objectif consistait à ajouter une fonction permettant de lire et stocker un automate fini à partir d'un fichier `.dot`. L'un des défis majeurs fut l'extraction fiable des informations pertinentes de ce format texte.

Nous avons implémenté la fonction `load_automaton_from_dot`, qui constitue le cœur de notre analyseur syntaxique (*parser*). Son rôle est de parcourir le fichier texte et de traduire les descriptions graphiques en données exploitables en mémoire. Afin de la rendre robuste et de contourner les spécificités du format DOT, la fonction opère selon les étapes séquentielles suivantes :

1. **Ouverture sécurisée du fichier** : Le fichier est ouvert en mode lecture (`"r"`). Si le fichier est introuvable ou illisible, le programme signale une erreur dans la console et interrompt le chargement proprement, évitant ainsi une erreur de segmentation.
2. **Filtrage ligne par ligne** : Le fichier est lu ligne après ligne via la fonction `fgets`. Pour optimiser le traitement, le programme ignore les lignes de décoration globale et se concentre exclusivement sur les lignes définissant des relations, identifiables par la présence de la chaîne d'arête `->`.
3. **Détection de l'état initial (Astuce du nœud fantôme)** : Dans les conventions de tracé GraphviZ, un état initial est souvent matérialisé par une flèche provenant d'un nœud invisible ou vide (par exemple : `"" -> 0`). Notre algorithme intercepte ce motif vide `""`, le remplace dynamiquement en mémoire par un identifiant lisible (`"IN"`), ce qui permet à la fonction `sscanf` de l'analyser. L'état pointé par cette flèche voit alors son attribut `is_initial` basculer à `true`.
4. **Détection des états finaux** : De manière similaire, notre code gère une autre convention où un état final pointe vers un nœud d'arrêt (par exemple : `3 -> fin_node`). Si la destination de la flèche commence par les lettres `"fin"`, l'état d'origine est enregistré comme étant un état acceptant (`is_final = true`).
5. **Extraction des transitions régulières et construction de l'alphabet** : Pour les véritables transitions entre deux états de l'automate :
   — Le programme isole les identifiants de l'état de départ et de l'état d'arrivée, en prenant soin de nettoyer les éventuels caractères parasites de syntaxe (tels que les crochets d'attributs `[` ou les points-virgules de fin de ligne `;`).
   — Il recherche ensuite l'attribut `label=` pour en extraire le symbole de transition, en gérant de manière sécurisée la lecture entre les guillemets.
   — Ces informations sont structurées sous forme d'une `Transition` puis ajoutées au tableau `transitions` de notre automate.

— *Gestion de l'alphabet :* Si l'étiquette extraite n'est pas le mot vide (représenté ici par la chaîne `"epsilon"`), elle est envoyée à la fonction `add_to_alphabet` pour enrichir dynamiquement l'alphabet global de l'automate sans créer de doublons.

6. **Sécurité par défaut (*Fallback* initial)** : Une fois le fichier entièrement lu et refermé, le programme effectue une ultime vérification. Si le formatage du fichier DOT ne précisait explicitement aucun état initial, l'algorithme affecte cette propriété par défaut au premier nœud identifié (indice 0), garantissant ainsi que l'automate reste valide et manipulable pour les opérations futures.

```c
bool load_automaton_from_dot(Automaton *automate, const char *filename)
{
  FILE *file = fopen(filename, "r");
  if (!file)
  {
    printf("Erreur: Impossible d'ouvrir le fichier %s\n", filename);
    return false;
  }

  char line[256];

  while (fgets(line, sizeof(line), file))
  {
    char *arrow_pos = strstr(line, "->");

    // Nous nous intéressons uniquement aux lignes contenant des transitions
    if (arrow_pos)
    {
      char from_str[64] = {0};
      char to_str[64] = {0};
      char label_str[64] = "epsilon";

      // Solution pour les nœuds avec chaîne vide (ex: "" -> 0)
      // Remplacer "" par un identifiant temporaire "IN" afin que sscanf puisse l'analyser comme un seul token
      char temp_line[256];
      strcpy(temp_line, line);
      char *empty_node = strstr(temp_line, "\"\"");
      if (empty_node && empty_node < (temp_line + (arrow_pos - line)))
      {
        empty_node[0] = 'I';
        empty_node[1] = 'N'; // Transforme "" en IN
      }

      // Extraire la source et la destination
      if (sscanf(temp_line, " %s -> %s", from_str, to_str) != 2)
        continue;

      // Supprimer les crochets ou points-virgules à la fin de la destination
      char *bracket = strchr(to_str, '[');
      if (bracket)
        *bracket = '\0';
      char *semi = strchr(to_str, ';');
      if (semi)
        *semi = '\0';

```

```
46      // 1. Détection de l'état initial ("IN" -> État)
47      if (strcmp(from_str, "IN") == 0)
48      {
49        int to_id = parse_etat_id(to_str);
50        if (to_id != -1)
51        {
52          int idx = get_or_create_etat_idx(automate, to_id);
53          automate->is_initial[idx] = true;
54        }
55        continue;
56      }
57
58      // 2. Détection de l'état final (État -> "fin*")
59      if (strncmp(to_str, "fin", 3) == 0)
60      {
61        int from_id = parse_etat_id(from_str);
62        if (from_id != -1)
63        {
64          int idx = get_or_create_etat_idx(automate, from_id);
65          automate->is_final[idx] = true;
66        }
67        continue;
68      }
69
70      // 3. Traitement des transitions normales
71      int from_id = parse_etat_id(from_str);
72      int to_id = parse_etat_id(to_str);
73
74      if (from_id != -1 && to_id != -1)
75      {
76        char *label_pos = strstr(line, "label");
77        if (label_pos)
78        {
79          char *quote1 = strchr(label_pos, '"');
80          if (quote1)
81          {
82            char *quote2 = strchr(quote1 + 1, '"');
83            if (quote2)
84            {
85              strncpy(label_str, quote1 + 1, quote2 - quote1 - 1);
86              label_str[quote2 - quote1 - 1] = '\0';
87            }
88          }
89          else
90          {
91            sscanf(label_pos, "label=%63[^ \t;,\]]", label_str);
92          }
93        }
94
95        int from_idx = get_or_create_etat_idx(automate, from_id);
96        int to_idx = get_or_create_etat_idx(automate, to_id);
97
98        if (automate->num_transitions < MAX_TRANSITIONS)
99        {
100         Transition *t = &automate->transitions[automate->
    num_transitions++];
101         t->from_etat = automate->etats[from_idx];
102         t->to_etat = automate->etats[to_idx];
```

```
            strcpy(t->label, label_str);

            if (strcmp(label_str, "epsilon") != 0)
            {
              add_to_alphabet(automate, label_str);
            }
          }
        }
      }
    }
  }
  // Par défaut, considérer le premier état (index 0) comme initial
  // si aucun état initial n'est explicitement défini via "" -> X
  bool has_initial = false;
  for (int i = 0; i < automate->num_etats; i++)
  {
    if (automate->is_initial[i])
    {
      has_initial = true;
      break;
    }
  }
  if (!has_initial && automate->num_etats > 0)
  {
    automate->is_initial[0] = true;
  }

  fclose(file);
  return true;
}
```

Listing 3 – Logique de la fonction `load_automaton_from_dot` (fichier `parser.c`)

5.3 Moteur d'affichage (`display.c`)

Une fois l'automate chargé, le cahier des charges exigeait d'ajouter une fonction permettant d'affichage les informations d'un automate passé en paramètre la liste des états, la liste des transitions, l'alphabet, les états finaux, les états initiaux. Ainsi, la fonction `display_automaton` a été développée.

Son rôle est de parcourir la structure en mémoire et d'afficher de façon lisible dans la console :

- La liste de tous les états récupérés.
- La liste restreinte des états initiaux et finaux (états acceptants).
- L'alphabet complet déduit des étiquettes des transitions.
- La liste complète des transitions formatée (ex : $\delta(q_0, a) = q_1$).

5.4 Interface interactive (`main.c`)

Pour orchestrer ces différentes fonctionnalités, nous devions ajouter un menu à notre programme. Ce menu a été intégré au fichier `main.c` et propose les options suivantes de manière cyclique :

1. **Charger un automate :** L'utilisateur saisit le chemin du fichier `.dot`.
2. **Afficher les informations :** Invoque la fonction `display_automaton`.
3. **Quitter :** Libère la mémoire et termine le programme proprement.

6 Difficultés Rencontrées et Solutions

Le développement de cette première itération n'a pas été sans obstacles. Nous soulignons notamment trois problématiques majeures résolues par l'équipe :

1. **Manipulation ardue des chaînes de caractères en C** : L'extraction fiable des labels de transition encadrés par des guillemets dans les fichiers DOT a nécessité des manipulations de pointeurs complexes et une forte vigilance quant aux débordements de tampons (buffers overflow), le C ne possédant pas de regex natives.
2. **Courbe d'apprentissage de Git** : Dans les premiers jours, la synchronisation du travail a été parasitée par des conflits de fusion (*merge conflicts*). Cela a imposé un temps d'adaptation et des sessions de résolution de conflits didactiques entre les membres.
3. **Incompatibilité OS pour la compilation** : La rigidité de notre premier `Makefile` n'a pas survécu à l'environnement Windows d'une partie de l'équipe, paralysant momentanément la compilation avant que nous n'adoptions l'approche *cross-platform* de CMake.
4. **Lecture des chemins de fichiers sous Windows** : Nous avons rencontré un problème lors de la saisie des chemins de fichiers avec la fonction `scanf("%255s", filename)`. Sous Windows, les chemins contiennent souvent des espaces et des antislahs qui faisaient échouer la lecture simple par cette fonction, nécessitant des ajustements pour capturer correctement la chaîne de caractères.

7 Conclusion de la Phase 1

En conclusion pour cette première semaine de développement, les bases techniques et organisationnelles du projet ont été solidement établies. La modélisation mathématique de l'automate fini a été transposée efficacement en structures C opérationnelles. Le système est désormais capable de dialoguer avec des fichiers standards (DOT) et de restituer l'information extraite, validant ainsi la totalité de la Partie 1 et ouvrant la voie à des manipulations plus complexes (opérations sur les automates, déterminisation, minimisation, construction de la table des symboles).