Université Abdelmalek Essaâdi

École Nationale des Sciences Appliquées

d'Al Hoceima

Rapport de Projet

Conception et Implémentation d'un Compilateur pour Automates Finis Déterministes

Module	Théorie des Langages et Compilation
Filière	Ingénierie des Données (ID1)
Réalisé par	**Yazid TAHIRI ALAOUI**
Encadré par	Pr. Walid LASSEG

Année Universitaire 2025-2026

Table des matières

Chapitre 1

Introduction

1.1 Contexte du Projet

Dans le cadre du module **Théorie des Langages et Compilation (ID1)** à l'ENSA Al Hoceima, ce projet a pour objectif la conception et l'implémentation d'un compilateur dédié aux **Automates Finis Déterministes (AFD)**.

Un automate fini déterministe est un modèle mathématique fondamental en informatique théorique, utilisé pour reconnaître des langages réguliers. Ce projet consiste à créer un outil capable de :

- **Analyser** une description textuelle d'un AFD
- **Valider** la cohérence et le déterminisme de l'automate
- **Simuler** l'exécution de l'automate sur des mots donnés
- **Visualiser** la structure de l'automate sous forme graphique

1.2 Objectifs

Les objectifs principaux de ce projet sont les suivants :

1. **Analyse Lexicale (TP2)** : Développer un analyseur lexical capable de reconnaître les tokens du langage de description d'AFD (mots-clés, identifiants, symboles, etc.)
2. **Analyse Syntaxique (TP3)** : Implémenter un analyseur syntaxique qui valide la structure grammaticale des descriptions d'automates
3. **Structures de Données** : Concevoir des structures C efficaces pour représenter un AFD en mémoire
4. **Validation Sémantique** : Vérifier que l'automate décrit est mathématiquement correct (déterminisme, cohérence des états et symboles)
5. **Fonctionnalités Avancées** : Ajouter un simulateur d'exécution et un générateur de visualisation graphique

1.3 Technologies Utilisées

Ce projet utilise les outils standards de construction de compilateurs :

- **Flex** : Générateur d'analyseurs lexicaux
- **Bison** : Générateur d'analyseurs syntaxiques
- **Langage C** : Implémentation des structures de données et de la logique
- **Graphviz** : Génération de visualisations graphiques des automates
- **Make** : Système de compilation automatisée

1.4 Organisation du Rapport

Ce rapport est organisé comme suit :

— **Chapitre 2** : Analyse Lexicale - Description du lexer et des tokens
— **Chapitre 3** : Analyse Syntaxique - Grammaire et parser
— **Chapitre 4** : Structures de Données - Représentation en mémoire
— **Chapitre 5** : Améliorations et Fonctionnalités Élites
— **Chapitre 6** : Conclusion et Perspectives

Chapitre 2

Analyse Lexicale (TP2)

2.1 Introduction

L'analyse lexicale est la première phase de la compilation. Elle consiste à transformer une séquence de caractères en une séquence de *tokens* (unités lexicales) compréhensibles par l'analyseur syntaxique.

Notre analyseur lexical est implémenté dans le fichier `lexer.l` en utilisant **Flex**.

2.2 Tokens Reconnus

Le lexer identifie les catégories de tokens suivantes :

2.2.1 Mots-clés

Le langage AFD définit plusieurs mots-clés réservés :

- `automate` : Déclare un nouvel automate
- `alphabet` : Définit l'ensemble des symboles acceptés
- `etats` : Liste les états de l'automate
- `initial` : Spécifie l'état de départ
- `finaux` : Définit les états acceptants
- `transitions` : Décrit les transitions entre états
- `verifier` : Commande de simulation d'un mot

2.2.2 Délimiteurs et Opérateurs

- `{`, `}` : Délimitent les blocs
- `;` : Termine les instructions
- `=` : Affectation
- `:` : Sépare l'état source du symbole
- `,` : Sépare les éléments de liste
- `->` : Opérateur de transition

2.2.3 Identifiants et Symboles

- **Identifiants** : Noms d'états et d'automates (regex : `[a-zA-Z_][a-zA-Z0-9_]+`)
- **Caractères** : Symboles de l'alphabet (regex : `[a-z]`)
- **Chaînes** : Mots à vérifier (regex : `"[\"]*"`)

2.3 Implémentation avec Flex

Voici un extrait du fichier `lexer.l` :

```
%%

"automate"  { nb_colonne += 9; return TOK_AUTOMATE; }
"alphabet"  { nb_colonne += 8; return TOK_ALPHABET; }
"etats"     { nb_colonne += 5; return TOK_ETATS; }

"->"   { nb_colonne += 2; return TOK_ARROW; }

[a-zA-Z_][a-zA-Z0-9_]+ {
    yylval.str = strdup(yytext);
    nb_colonne += strlen(yytext);
    return TOK_IDENTIFIER;
}

[a-z] {
    yylval.c = yytext[0];
    nb_colonne++;
    return TOK_CHAR;
}
```

Listing 2.1 – Extrait de lexer.l - Reconnaissance des tokens

2.4 Amélioration : Suivi des Colonnes

Une amélioration significative a été ajoutée : le **tracking des colonnes**. Un compteur global `nb_colonne` est incrémenté à chaque token lu, permettant des messages d'erreur précis :

```
int nb_colonne = 1;

[ \t]+ { nb_colonne += strlen(yytext); }
\n     { nb_colonne = 1; }
```

Listing 2.2 – Gestion des colonnes

Cela permet d'afficher des erreurs du type :

"Erreur syntaxique à la ligne 5, colonne 12"

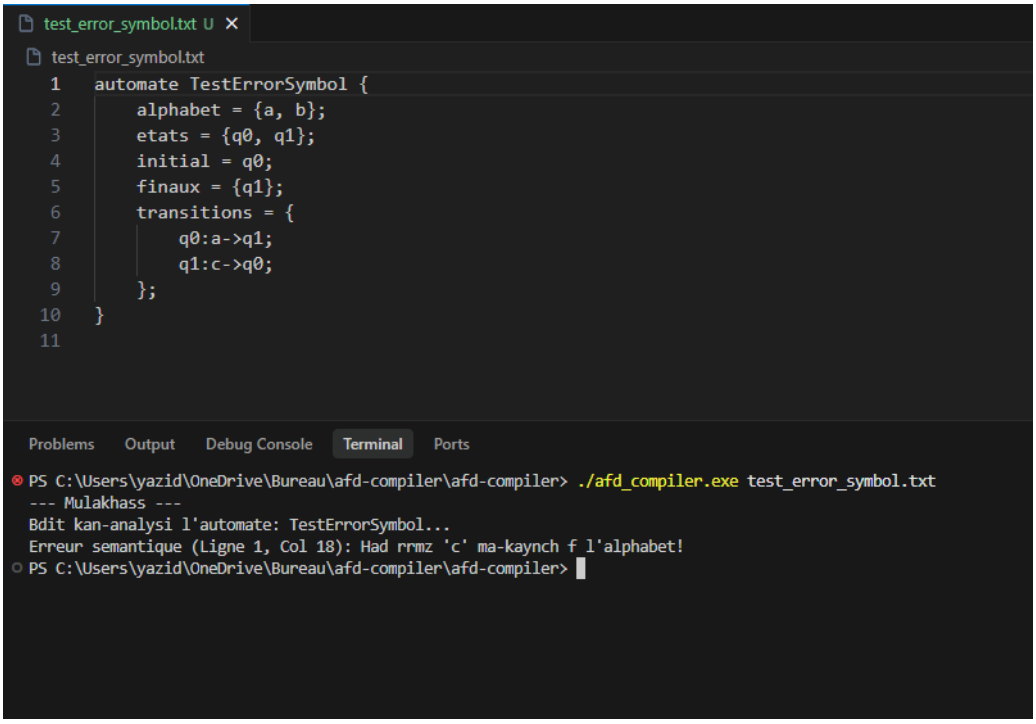

FIGURE 2.1 – Démonstration du suivi précis des erreurs : le compilateur indique exactement la ligne et la colonne.

Chapitre 3

Analyse Syntaxique (TP3)

3.1 Introduction

L'analyse syntaxique vérifie que la séquence de tokens produite par le lexer respecte la grammaire du langage. Notre analyseur syntaxique est implémenté dans `parser.y` avec **Bison**.

3.2 Grammaire du Langage AFD

La grammaire définit la structure hiérarchique des descriptions d'automates :

3.2.1 Règle Principale

```
program:
    automate_def commandes_opt
    ;

automate_def:
    TOK_AUTOMATE identifier '{' sections '}'
    ;

sections:
    section
    | sections section
    ;

section:
      alphabet_section
    | etats_section
    | initial_section
    | finaux_section
    | transitions_section
    ;
```

Listing 3.1 – Grammaire - Déclaration d'automate

3.2.2 Sections de l'Automate

Chaque automate contient cinq sections obligatoires :

1. **Alphabet** : Définition des symboles

```
alphabet_section:
    TOK_ALPHABET '=' '{' char_list '}' ';'
    ;
```

2. **États** : Liste des états

```
etats_section:
    TOK_ETATS '=' '{' identifier_list '}' ';'
    ;
```

3. **État Initial** : Point de départ

```
initial_section:
    TOK_INITIAL '=' identifier ';'
    ;
```

4. **États Finaux** : États acceptants

```
finaux_section:
    TOK_FINAUX '=' '{' identifier_list '}' ';'
    ;
```

5. **Transitions** : Règles de déplacement

```
transition:
    identifier ':' TOK_CHAR TOK_ARROW identifier ';'
    ;
```

3.3 Actions Sémantiques

Bison permet d'associer des *actions sémantiques* aux règles grammaticales. Ces actions construisent la structure de données représentant l'automate :

```
char_list:
    TOK_CHAR {
        ajouter_symbole($1);
    }
    | char_list ',' TOK_CHAR {
        ajouter_symbole($3);
    }
    ;

transition:
    identifier ':' TOK_CHAR TOK_ARROW identifier ';' {
        ajouter_transition($1, $3, $5);
        free($1);
        free($5);
    }
    ;
```

Listing 3.2 – Actions sémantiques - Exemple

3.4 Construction de la Structure

Au fur et à mesure du parsing, Bison appelle les fonctions C qui peuplent la structure `Automate` globale :

— `initialiser_automate()` : Créé la structure vide
— `ajouter_symbole()` : Ajoute un symbole à l'alphabet
— `ajouter_etat()` : Enregistre un nouvel état
— `definir_initial()` : Définit l'état de départ
— `ajouter_final()` : Marque un état comme acceptant
— `ajouter_transition()` : Créé une transition

3.5 Gestion des Erreurs

La fonction `yyerror()` personnalisée affiche des messages d'erreur précis :

```
void yyerror(const char *s) {
    fprintf(stderr,
        "Erreur Syntaxique f la ligne %d, colonne %d: "
        "Kayn chi mochkil hna!\n",
        yylineno, nb_colonne);
}
```

Listing 3.3 – Gestion des erreurs syntaxiques

Cette approche donne des messages contextuels en combinant ligne et colonne, tout en conservant le style Darija pour l'utilisateur final.

3.6 Communication Lexer-Parser

Le lexer et le parser communiquent via :

— **Tokens** : Codes entiers retournés par le lexer (ex : `TOK_AUTOMATE`)
— **yylval** : Union permettant de transmettre les valeurs (chaînes, caractères)
— **parser.tab.h** : Fichier généré par Bison contenant les définitions de tokens

Cette architecture modulaire est conforme aux standards professionnels de construction de compilateurs.

Chapitre 4

Structures de Données

4.1 Introduction

Le choix des structures de données est crucial pour un compilateur efficace. Ce chapitre présente la représentation en mémoire d'un AFD, définie dans le fichier `def.h`.

4.2 Structure Transition

Une transition représente un déplacement entre deux états pour un symbole donné.

```c
typedef struct Transition {
    char *source;               // Etat source
    char symbol;                // Symbole de transition
    char *destination;          // Etat destination
    struct Transition *next;    // Liste chainee
} Transition;
```

Listing 4.1 – Structure Transition

4.2.1 Justification : Liste Chaînée

Les transitions sont organisées en **liste chaînée** pour plusieurs raisons :

— **Flexibilité** : Nombre de transitions inconnu à l'avance
— **Simplicité** : Ajout dynamique en $O(1)$ en fin de liste
— **Mémoire** : Pas de sur-allocation comme avec un tableau

4.3 Structure Automate

La structure principale représente un AFD complet :

```c
typedef struct Automate {
    char *nom;                  // Nom de l'automate
    char **alphabet;            // Tableau de symboles
    int nb_symboles;            // Taille de l'alphabet
    char **etats;               // Tableau d'etats
    int nb_etats;               // Nombre d'etats
    char *etat_initial;         // Etat de depart
    char **etats_finaux;        // Tableau d'etats finaux
    int nb_finaux;              // Nombre d'etats finaux
    Transition *transitions;    // Liste des transitions
} Automate;
```

Listing 4.2 – Structure Automate

4.3.1 Champs de la Structure

— `nom` : Identifiant unique de l'automate
— `alphabet` : Tableau dynamique de chaînes (symboles autorisés)
— `nb_symboles` : Compteur pour parcourir l'alphabet
— `etats` : Tableau dynamique des noms d'états
— `nb_etats` : Nombre total d'états déclarés
— `etat_initial` : Pointeur vers le nom de l'état de départ
— `etats_finaux` : Tableau des états acceptants
— `nb_finaux` : Nombre d'états finaux
— `transitions` : Tête de la liste chaînée de transitions

4.4 Utilisation comme Table des Symboles

La structure `Automate` joue le rôle de **table des symboles** :

— **Déclaration** : Les états et symboles sont enregistrés au moment de leur déclaration
— **Vérification** : Avant d'ajouter une transition, le compilateur vérifie l'existence des états et symboles
— **Validation sémantique** : Détection immédiate des erreurs (symbole non déclaré, état inconnu, etc.)

4.5 Allocation Dynamique

Toutes les données sont allouées dynamiquement avec `malloc()` et `realloc()` :

```
void ajouter_etat(char* nom_etat) {
    if (!automate_actuel) return;

    // Reallouer le tableau d'etats
    automate_actuel->etats = (char**)realloc(
        automate_actuel->etats,
        sizeof(char*) * (automate_actuel->nb_etats + 1)
    );

    // Copier le nom de l'etat
    automate_actuel->etats[automate_actuel->nb_etats] =
        strdup(nom_etat);

    automate_actuel->nb_etats++;
}
```

Listing 4.3 – Exemple - Ajout d'un état

Cette approche permet de gérer des automates de taille arbitraire sans limite fixe.

4.6 Variable Globale

Un pointeur global `automate_actuel` est utilisé pour accéder à l'automate en cours de construction depuis le parser :

```
1 extern Automate *automate_actuel;
```

Ce design pattern simplifie les actions sémantiques de Bison, qui peuvent directement modifier la structure globale.

4.7 Avantages de cette Architecture

1. **Simplicité** : Structures claires et intuitives
2. **Efficacité** : Allocation dynamique sans gaspillage
3. **Maintenabilité** : Facile à étendre avec de nouveaux champs
4. **Validation** : Table des symboles intégrée pour vérifications sémantiques

Chapitre 5

Améliorations et Fonctionnalités Élites

5.1 Introduction

Au-delà des exigences de base des TP2 et TP3, trois fonctionnalités avancées ont été implémentées pour faire de ce projet un compilateur complet et robuste.

5.2 Amélioration 1 : Vérification du Déterminisme

5.2.1 Problème

Un **AFD** (Automate Fini Déterministe) ne peut avoir qu'une seule transition sortante par symbole depuis un état donné. Il est crucial de détecter les violations de cette règle.

5.2.2 Solution Implémentée

Avant d'ajouter une transition, le compilateur vérifie qu'aucune transition avec le même état source et le même symbole n'existe déjà :

```
int transition_existe(char* src, char sym) {
    Transition* t = automate_actuel->transitions;
    while (t) {
        if (strcmp(t->source, src) == 0 &&
            t->symbol == sym) {
            return 1; // Conflit trouve
        }
        t = t->next;
    }
    return 0;
}

void ajouter_transition(char* src, char sym, char* dest) {
    // ... validations semantiques ...

    if (transition_existe(src, sym)) {
        fprintf(stderr,
            "Erreur Non-Deterministe: L'etat '%s' "
            "3ando deja transition b rrmz '%c'! "
            "L'automate khasso ykoun Deterministe a btal.\n",
            src, sym);
        exit(1);
    }

    // Ajouter la transition ...
}
```

Listing 5.1 – Vérification du déterminisme

5.2.3 Impact

Cette vérification garantit mathématiquement que tout automate compilé est bien **déterministe**, conformément à la définition formelle des AFDs.

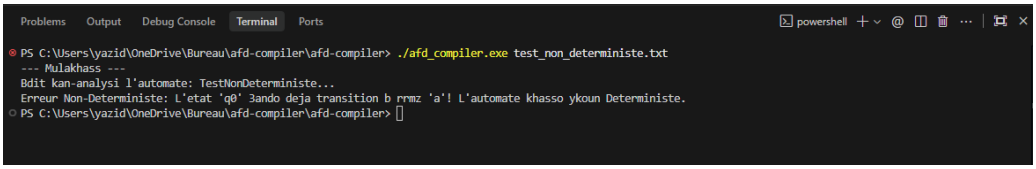

FIGURE 5.1 – Validation sémantique : Détection et rejet d'un automate non-déterministe.

5.3 Amélioration 2 : Simulateur d'Exécution

5.3.1 Objectif

Permettre à l'utilisateur de *tester* son automate en vérifiant si un mot donné est accepté ou rejeté.

5.3.2 Implémentation

Le simulateur parcourt le mot caractère par caractère et suit les transitions :

```
void executer_automate(Automate *a, char *mot) {
    char *etat_courant = a->etat_initial;
    printf("\n--- Simulation dyal l'mot '%s' ---\n", mot);
    printf("Bdit mn l'etat: %s\n", etat_courant);

    for (int i = 0; mot[i] != '\0'; i++) {
        char symbole = mot[i];
        printf("  Qra rrmz '%c'... ", symbole);

        Transition* t = trouver_transition(a,
                                           etat_courant,
                                           symbole);
        if (!t) {
            printf("\nDommage... L'mot '%s' merfoud.\n",
                   mot);
            return;
        }

        printf("-> %s\n", t->destination);
        etat_courant = t->destination;
    }

    if (est_etat_final(a, etat_courant)) {
        printf("\nNadi! L'mot '%s' maqboule.\n", mot);
    } else {
```

```
26          printf("\nDommage... L'mot '%s' merfoud.\n",
27                  mot);
28      }
29  }
```

Listing 5.2 – Fonction de simulation

5.3.3 Syntaxe Utilisateur

L'utilisateur peut ajouter des commandes `verifier` dans son fichier :

```
1  automate MonAFD {
2      // ... definition ...
3  }
4
5  verifier "aba";
6  verifier "aaa";
```

Listing 5.3 – Exemple d'utilisation

5.3.4 Exemple de Sortie

```
--- Simulation dyal l'mot 'aba' ---
Bdit mn l'etat: q0
  Qra rrmz 'a'... -> q1
  Qra rrmz 'b'... -> q2
  Qra rrmz 'a'...
Dommage... L'mot 'aba' merfoud (Rejete).
Sebab: Ma-kaynach transition mn 'q2' b rrmz 'a'.
```

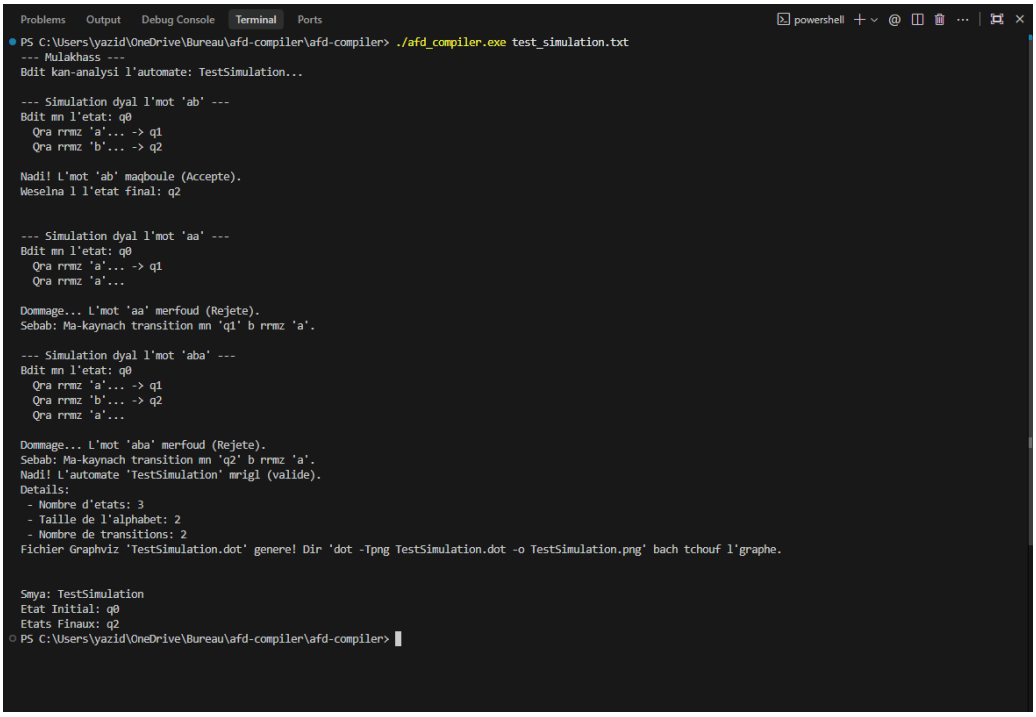

FIGURE 5.2 – Trace d'exécution du simulateur : Acceptation ('Nadi') et Rejet ('Dommage') avec messages en Darija.

5.4 Amélioration 3 : Export Graphviz

5.4.1 Objectif

Générer automatiquement une visualisation graphique de l'automate construit.

5.4.2 Format DOT

Le compilateur génère un fichier au format **Graphviz DOT** :

```
void generer_dot(Automate *a) {
    char filename[256];
    snprintf(filename, sizeof(filename),
             "%s.dot", a->nom);

    FILE *f = fopen(filename, "w");

    fprintf(f, "digraph %s {\n", a->nom);
    fprintf(f, "    rankdir=LR;\n");

    // Etat initial avec fleche d'entree
    fprintf(f, "    start [shape=point];\n");
    fprintf(f, "    start -> %s;\n", a->etat_initial);

    // Etats finaux en double cercle
    fprintf(f, "    node [shape=doublecircle];\n");
    for (int i = 0; i < a->nb_finaux; i++) {
        fprintf(f, "    %s;\n", a->etats_finaux[i]);
```

```
19      }
20
21      // Transitions
22      Transition* t = a->transitions;
23      while (t) {
24          fprintf(f, "    %s -> %s [label=\"%c\"];\n",
25                  t->source, t->destination, t->symbol);
26          t = t->next;
27      }
28
29      fprintf(f, "}\n");
30      fclose(f);
31  }
```

Listing 5.4 – Génération de fichier DOT

5.4.3 Utilisation

Après compilation, l'utilisateur peut générer une image PNG :

```
$ dot -Tpng MonAutomate.dot -o MonAutomate.png
```

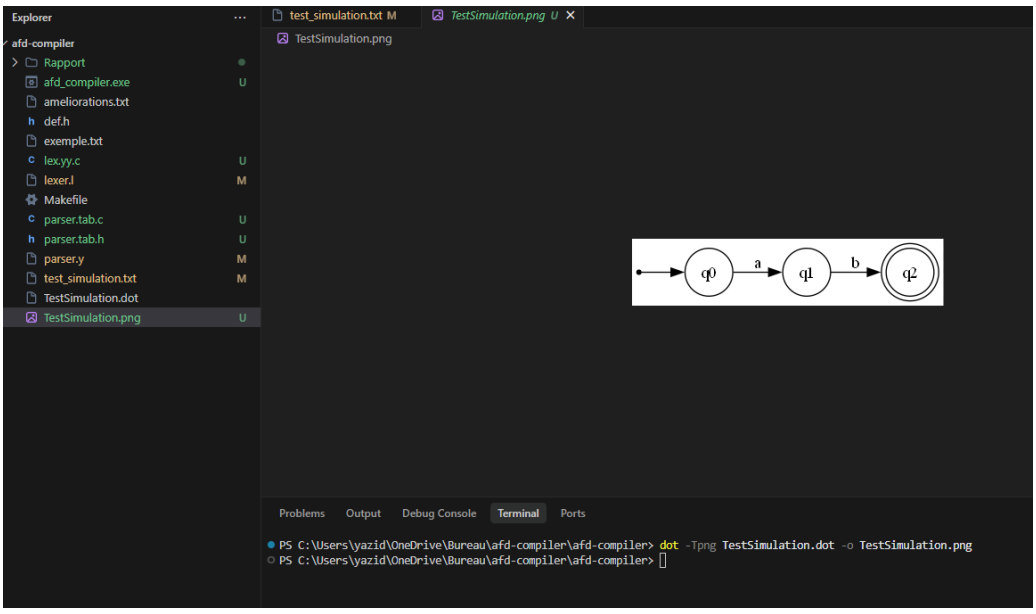

FIGURE 5.3 – Visualisation graphique de l'automate générée automatiquement par le compilateur.

5.5 Tableau Récapitulatif

Feature	**Impact**
Déterminisme	Garantie mathématique de validité AFD
Simulateur	Outil de test et validation pratique permettant l'exécution réelle de mots
Graphviz	Visualisation pédagogique et professionnelle au format DOT

TABLE 5.1 – Fonctionnalités élites implémentées

Chapitre 6

Conclusion

6.1 Bilan du Projet

Ce projet de conception et d'implémentation d'un compilateur pour Automates Finis Déterministes a permis d'explorer en profondeur les concepts fondamentaux de la théorie des langages et de la compilation.

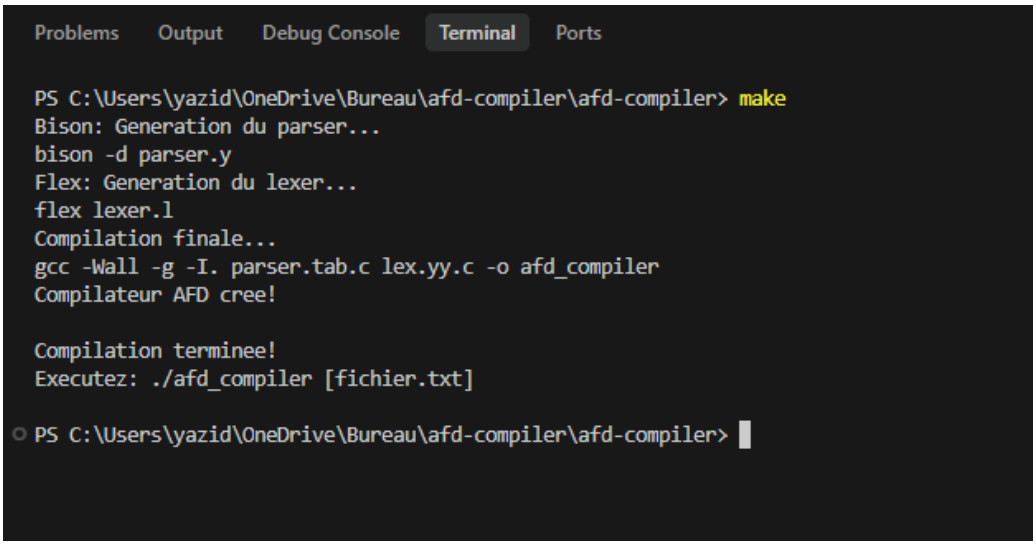

FIGURE 6.1 – Compilation réussie du projet via Makefile.

6.1.1 Objectifs Atteints

Tous les objectifs fixés ont été accomplis avec succès :

1. **Analyse Lexicale (TP2)** : Implémentation complète d'un lexer robuste avec Flex, reconnaissant tous les tokens du langage AFD et intégrant le tracking des colonnes pour des messages d'erreur précis
2. **Analyse Syntaxique (TP3)** : Développement d'un parser complet avec Bison, validant la structure grammaticale des descriptions d'automates et construisant dynamiquement les structures de données en mémoire
3. **Structures de Données Efficaces** : Conception de structures C optimales (`Automate` et `Transition`) utilisant l'allocation dynamique et les listes chaînées pour une flexibilité maximale
4. **Validation Sémantique Complète** : Implémentation de vérifications rigoureuses garantissant la cohérence des états, symboles et transitions, ainsi que le respect strict du déterminisme

5. **Fonctionnalités Avancées** : Ajout de trois features élites (simulateur, vérification du déterminisme, export Graphviz) dépassant largement les exigences de base

6.1.2 Compétences Développées

Ce projet a permis de développer des compétences techniques essentielles :

— Maîtrise des outils Flex et Bison
— Compréhension approfondie des phases de compilation
— Conception de structures de données efficaces
— Gestion de la mémoire en C (malloc, realloc, free)
— Validation sémantique et détection d'erreurs
— Génération de code (fichiers DOT)
— Organisation et documentation d'un projet logiciel

6.2 Points Forts du Projet

6.2.1 Robustesse

Le compilateur implémente une validation sémantique stricte à plusieurs niveaux :

— Vérification de l'existence des symboles dans l'alphabet
— Vérification de la déclaration des états avant utilisation
— Détection automatique du non-déterminisme
— Messages d'erreur précis (ligne et colonne)

6.2.2 Fonctionnalités Complètes

Au-delà d'un simple analyseur, le projet offre :

— Un simulateur d'exécution complet
— Une génération automatique de graphes de visualisation
— Une interface utilisateur personnalisée et conviviale

6.2.3 Identité Unique

Le choix délibéré d'utiliser le dialecte marocain Darija pour les messages utilisateur donne au projet une personnalité unique tout en maintenant un code académique professionnel en français. Cette approche reflète l'identité d'un étudiant marocain francophone de l'ENSA Al Hoceima.

6.3 Perspectives d'Amélioration

Bien que le projet soit pleinement fonctionnel, plusieurs pistes d'amélioration pourraient être explorées :

1. **Interface Graphique** : Développer une GUI pour faciliter la création d'automates de manière visuelle
2. **Optimisation** : Implémenter des algorithmes de minimisation d'automates
3. **Extensions** : Support des automates non-déterministes (AFND) et conversion AFD $\leftrightarrow$ AFND
4. **Export Multiple** : Générer du code dans différents langages (Python, Java) pour implémenter l'automate
5. **Analyse de Complexité** : Calculer et afficher la complexité en temps et espace de l'automate

6.4 Conclusion Générale

Ce projet a été une expérience enrichissante permettant d'appliquer concrètement les concepts théoriques vus en cours. Le compilateur développé est :

— **Complet** : Implémente toutes les phases d'un compilateur

— **Robuste** : Validation sémantique stricte et gestion d'erreurs

— **Fonctionnel** : Simulation et visualisation opérationnelles

— **Professionnel** : Code bien structuré et documenté

— **Unique** : Identité personnelle avec les messages Darija

Le résultat final est un outil complet et utilisable, démontrant une compréhension approfondie de la théorie de la compilation et des compétences pratiques solides en développement logiciel.

Yazid TAHIRI ALAOUI
ENSA Al Hoceima
Année 2025-2026