INSEA

المندوبية السامية للتخطيط
HAUT-COMMISSARIAT AU PLAN

Introduction :

Après avoir suivi le cours de Technique de Compilation, notre équipe s'est lancée dans un projet d'envergure visant à mettre en œuvre un programme en Java. L'objectif de ce projet consiste à élaborer un système capable de recevoir en entrée un automate à états finis, représenté sous la forme d'un graphe, et de générer en sortie sa représentation procédurale directe. Cette représentation sera ensuite exploitée pour l'analyse lexicale, une étape cruciale dans le processus de compilation.

En choisissant Java comme langage de programmation, reconnu pour sa flexibilité et sa puissance, notre ambition est de créer un outil robuste capable de manipuler efficacement les automates à états finis.

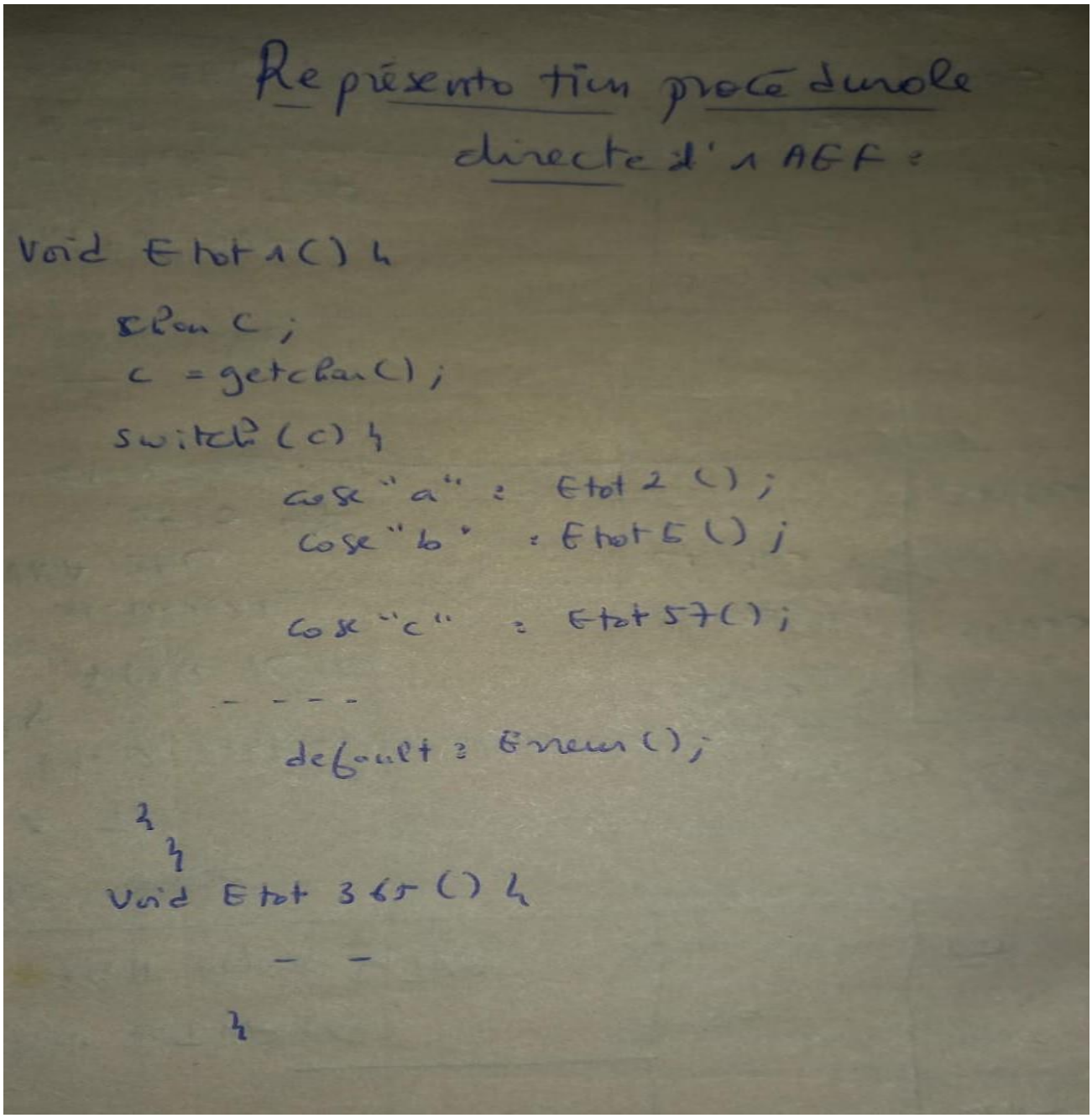
Concept du projet :

Ecrire un programme qui reçoit en entrée un automate à état fini en tant que graphe et produit en sortie sa représentation procédurale directe et l'utiliser pour l'analyse lexicale.

La réalisation du projet

Phase d’analyse :

Lors de la phase initiale de notre projet, notre binôme s'est réuni pour examiner attentivement les exigences et comprendre clairement les attentes du projet. Nous avons posé sur la table toutes les composantes nécessaires, cherchant à décomposer chaque élément du problème pour en saisir pleinement les subtilités.



Mots-Clés du Projet :

- ***Automate a état fini :***

Un automate à états finis (AEF), également appelé automate fini déterministe (AFD), est un modèle mathématique utilisé en informatique théorique pour représenter un système comportant un nombre fini d'états, des transitions entre ces états et un ensemble fini de symboles d'entrée. Il se compose d'un ensemble fini d'états, de règles de transition définissant le passage d'un état à un autre en réaction à des symboles d'entrée spécifiques, un état initial, et un ensemble d'états finaux déterminant où l'automate

termine son exécution. Les automates à états finis sont largement employés dans la modélisation de systèmes réactifs et sont particulièrement utiles pour la reconnaissance de langages réguliers et l'analyse lexicale dans le contexte de la compilation.

Sa représentation :

A = (T, E, D, ei, Ef) avec :

T = Ensemble des unités terminales

E = Ensemble des états possibles d'A

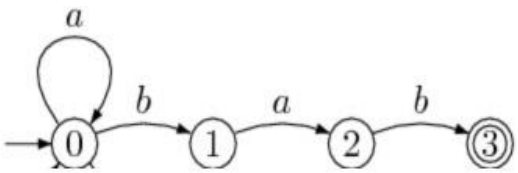
D = Ensemble des règles de transition

ei = Etat initial

Ef = Ensemble des états finaux

- ***Représentation procédurale directe d'un automate a états finis :***

Voici la représentation procédurale directe de cet automate :




```
void Etat 0 ()
{
        char c;
        c= getchar() ;
        switch(c) {
                cas ' a ' : Etat 0 ();
```

```
                    break;
            cas ' b ' : Etat 1 ();
                    break;
            default : Erreur() ;
        }
}

void Etat 1 ()
{
        char c;
        c= getchar() ;
        switch(c) {
            cas ' a ' : Etat 2 ();
             break;
            default : Erreur() ;
        }
}




void Etat 2 ()
{
        char c;
        c= getchar() ;
        switch(c) {
                cas ' b ' : Etat 3 ();
                        break;
                default : Erreur() ;
        }
}
```

```
void Etat 3 ()
{
        char c;
        c= getchar() ;
        switch(c) {
                default : Erreur() ;
        }
}
```

- ***Analyse lexicale :***

Dans le contexte d'un automate à états finis, l'analyse lexicale se réfère au processus d'interprétation d'une séquence de symboles ou de caractères en entrée conformément aux règles définies par l'automate. L'automate à états finis modélise généralement les patterns lexicaux du langage, et l'analyse lexicale implique la reconnaissance et l'identification de ces patterns pendant l'exécution.

Explication du code :

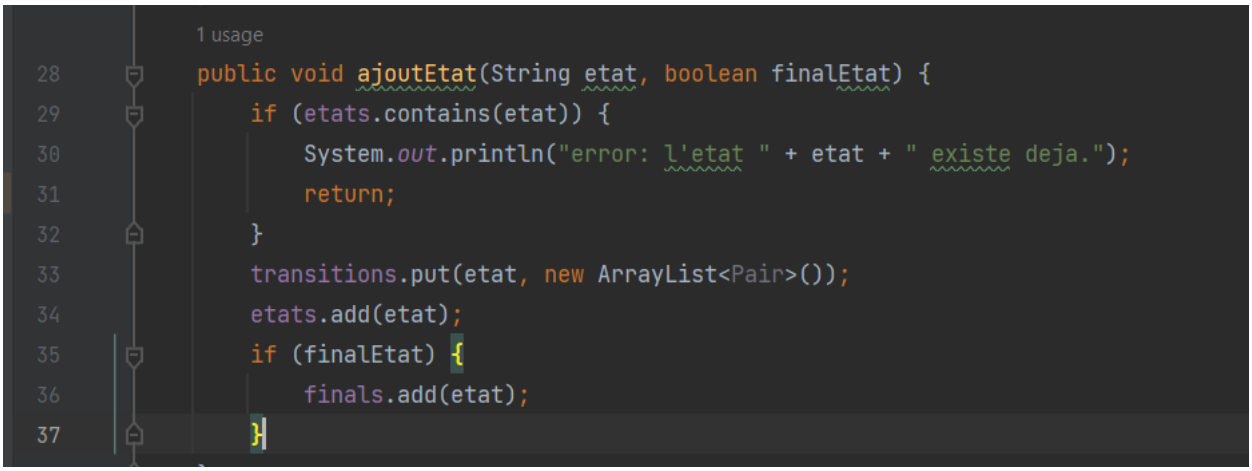
- Constructeur de la classe AEF :

Le constructeur de la classe **AEF** initialise les structures de données nécessaires pour représenter un automate fini déterministe (états, transitions, états finaux, états initiaux) et construit l'ensemble des étiquettes terminales en éliminant les doublons.

```
4 usages
14      private String UTer;
1 usage
15  @   public AEF(String UTer) {
16          this.etats = new ArrayList<>();
17          this.transitions = new HashMap<>();
18          this.init = new ArrayList<>();
19          this.finals = new ArrayList<>();
20          this.UTer = "";
21
22          for (char s : UTer.toCharArray()) {
23              if (this.UTer.indexOf(s) == -1) {
24                  this.UTer += s;
25              }
26          }
27      }
```

- Méthode ajouteEtat :

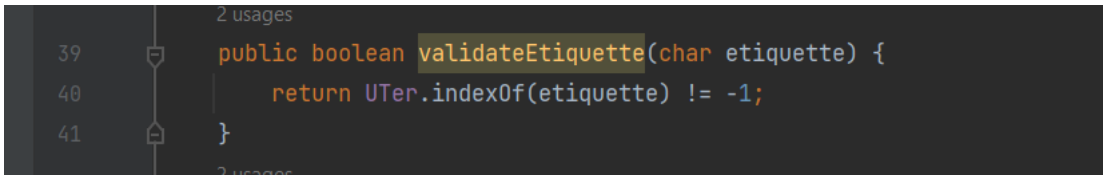
  La méthode **ajoutEtat** de la classe **AEF** ajoute un état à l'automate. Elle vérifie d'abord si l'état existe déjà, affiche une erreur le cas échéant, puis initialise la liste des transitions associée à cet état. Enfin, elle ajoute l'état à la liste globale des états et, s'il est final, à la liste des états finaux.



```
1 usage
28  public void ajoutEtat(String etat, boolean finalEtat) {
29      if (etats.contains(etat)) {
30          System.out.println("error: l'etat " + etat + " existe deja.");
31          return;
32      }
33      transitions.put(etat, new ArrayList<Pair>());
34      etats.add(etat);
35      if (finalEtat) {
36          finals.add(etat);
37      }
```

- Méthode validateEtiquette :

  La méthode **validateEtiquette** de la classe **AEF** valide si un symbole d'étiquette donné fait partie de l'ensemble d'étiquettes terminales (**UTer**).



```
2 usages
39  public boolean validateEtiquette(char etiquette) {
40      return UTer.indexOf(etiquette) != -1;
41  }
```

- Méthode getDestinationEtat :

  La méthode **getDestinationEtat** obtient l'état de destination d'une transition à partir d'un état source et d'un symbole d'étiquette donnés. Elle vérifie d'abord si l'état source existe, puis itère sur les transitions associées à cet état pour trouver la transition correspondant au symbole d'étiquette. Si la transition est trouvée, la méthode retourne l'état de destination; sinon, elle retourne **null**.

```java
2 usages
public String getDestinationEtat(String srcEtat, char etiquette) {
    if (!etats.contains(srcEtat)) {
        System.out.println("error: the state " + srcEtat + " is not an existing state.");
        return null;
    }

    for (Pair pair : transitions.get(srcEtat)) {
        if (pair.getEtiquette() == etiquette) {
            return pair.getDestinationEtat();
        }
    }
    return null;
}
```

- Méthode addTransition :

La méthode **addTransition** de la classe **AEF** ajoute une transition à l'automate entre un état source, un symbole d'étiquette, et un état de destination. Elle effectue plusieurs vérifications, notamment si le symbole d'étiquette est valide, si les états source et destination existent, et si la transition n'existe pas déjà. Si toutes les conditions sont remplies, la méthode crée une nouvelle transition et l'ajoute à la liste des transitions associée à l'état source.

```java
1 usage
public void addTransition(String srcEtat, char etiquette, String destEtat) {
    if (!validateEtiquette(etiquette)) {
        System.out.println("error: le symbole " + etiquette + " ne fait pas partie de T.");
        return;
    }

    if (!etats.contains(srcEtat)) {
        System.out.println("error: l'etat " + srcEtat + " n'existe pas.");
        return;
    }

    if (!etats.contains(destEtat)) {
        System.out.println("error: l'etat " + srcEtat + " n'existe pas.");
        return;
    }

    if (getDestinationEtat(srcEtat, etiquette) != null) {
        System.out.println("error: la transition(" + srcEtat + "," + etiquette + ",...) existe deja.");
        return;
    }
    transitions.get(srcEtat).add(new Pair(etiquette, destEtat));
}
1 usage
```

- Méthode affichageProcDirecte :

La méthode **affichageProcDirecte** génère du code source C en utilisant une structure de commutation pour chaque état de l'automate. Elle crée des fonctions procédurales directes pour chaque transition, utilisant une commutation basée sur l'étiquette de transition. En cas de symbole inattendu, elle appelle une fonction d'erreur.

```
1 usage
public void affichageProcDirecte() {
    for (String etat : etats) {
        System.out.println("void etat " + etat + "() {");
        System.out.println("    char c;");
        System.out.println("    c = getchar();");
        System.out.println("    switch(c) {");

        for (Pair pair : transitions.get(etat)) {
            System.out.println("        case '" + pair.getEtiquette() + "': etat " + pair.getDestinationEtat() + "(); break;");
        }

        System.out.println("        default: Erreur();");
        System.out.println("    }");
        System.out.println("}\n");
    }
}
```

- Méthode analyserLexical :

La méthode **analyserLexical** effectue l'analyse lexicale d'une chaîne d'entrée en utilisant l'automate à états finis défini. Elle parcourt chaque symbole de la chaîne, vérifie sa validité en tant que symbole terminal, puis utilise les transitions pour passer à l'état suivant. Si une transition est invalide, elle affiche une erreur. À la fin de l'analyse, elle vérifie si l'automate a atteint un état final, signalant ainsi le succès ou l'échec de l'analyse lexicale.

```
1 usage
public void analyserLexical(String input) {
    for (String initialState : init) {
        String currentEtat = initialState;

        for (char c : input.toCharArray()) {
            if (!validateEtiquette(c)) {
                System.out.println("Erreur: Le symbole '" + c + "' n'est pas un symbole terminal valide.");
                return;
            }

            String destinationEtat = getDestinationEtat(currentEtat, c);

            if (destinationEtat != null) {
                currentEtat = destinationEtat;
            } else {
                System.out.println("Erreur: Transition invalide pour le symbole '" + c + "' depuis l'état '" + currentEtat + "'.");
                return;
            }
        }

        if (finals.contains(currentEtat)) {
            System.out.println("Analyse lexicale réussie.");
            return; // Sortir de la méthode dès qu'une analyse réussie est trouvée
        }
```

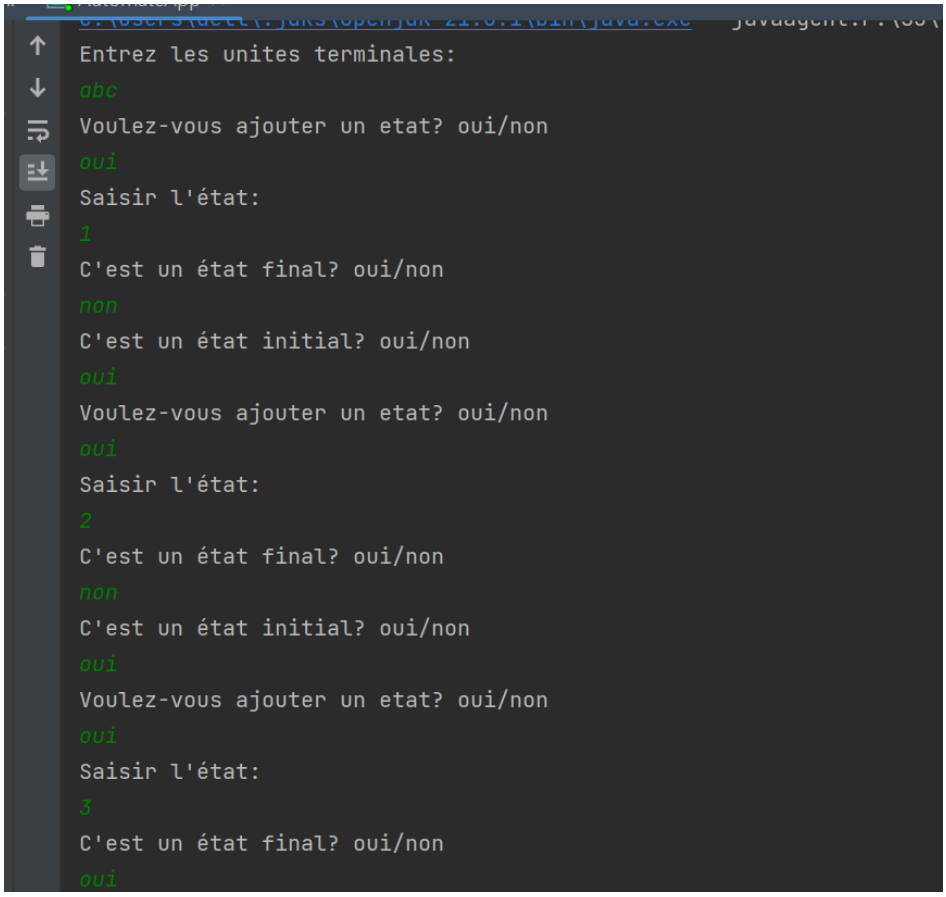
- Espace utilisateur :

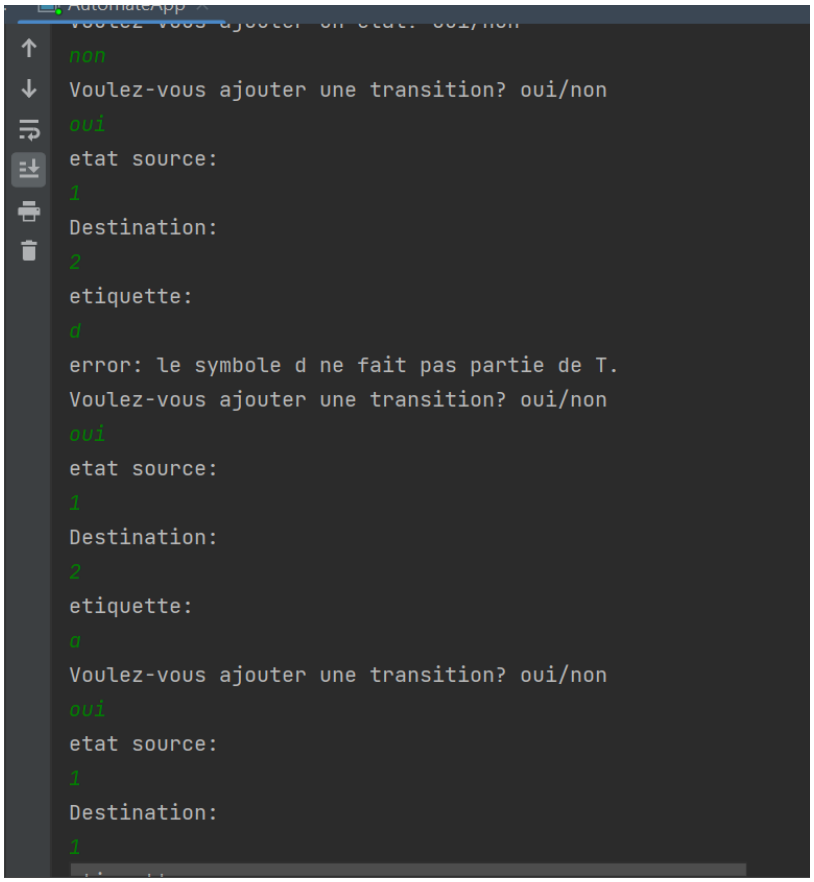
Demander à l'utilisateur d'entrer les données nécessaires pour l'automate.

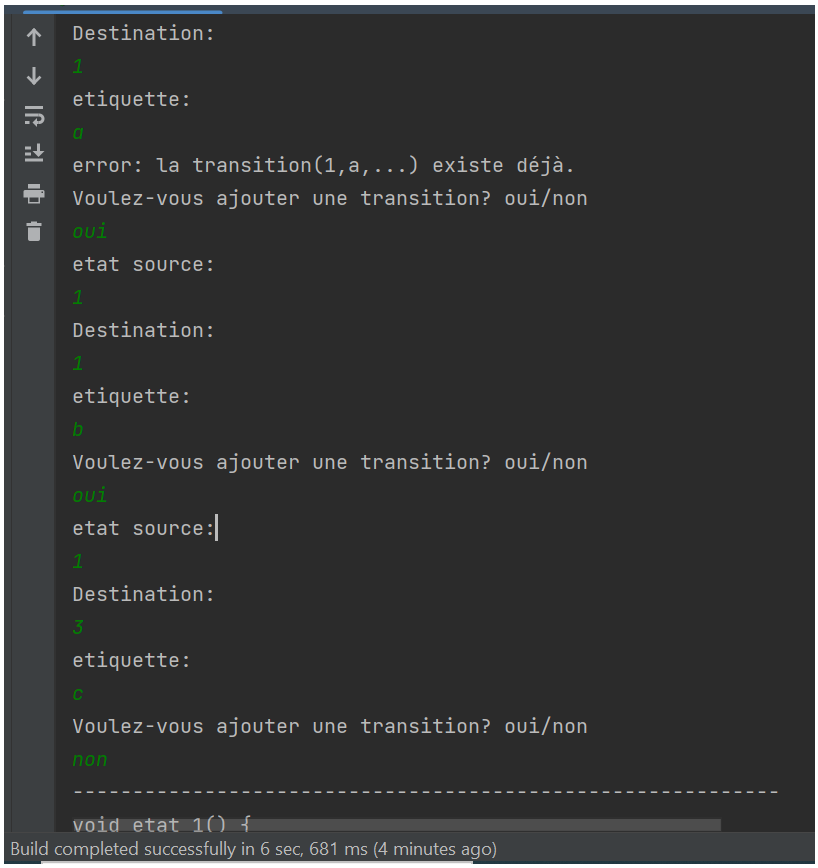
```java
        boolean continuer = true;

        // Initialiser l'automate une seule fois en dehors de la boucle
        System.out.println("Entrez les unites terminales:");
        String c = scanner.nextLine();
        AEF a = new AEF(c);

        while (continuer) {
            boolean ajouterEtats = true;
            boolean ajouterTransitions = true;

            while (ajouterEtats) {
                System.out.println("Voulez-vous ajouter un etat? oui/non");
                String i = scanner.nextLine();
                if ("oui".equalsIgnoreCase(i)) {
                    System.out.println("Saisir l'état:");
                    String state = scanner.nextLine();
                    System.out.println("C'est un état final? oui/non");
                    String isFinal = scanner.nextLine();
                    a.ajoutEtat(state, "oui".equalsIgnoreCase(isFinal));
                } else {
                    ajouterEtats = false;
                }
            }
```

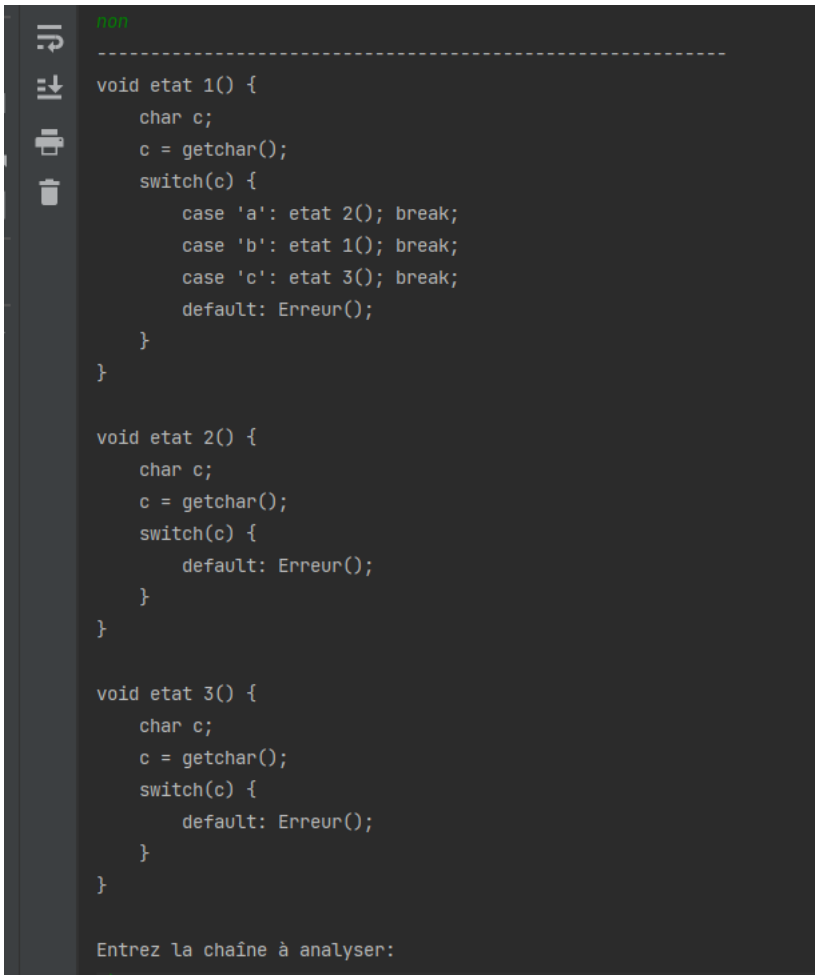
```java

            while (ajouterTransitions) {
                System.out.println("Voulez-vous ajouter une transition? oui/non");
                String i = scanner.nextLine();
                if ("oui".equalsIgnoreCase(i)) {
                    System.out.println("etat source:");
                    String srcState = scanner.nextLine();
                    System.out.println("Destination:");
                    String dstState = scanner.nextLine();
                    System.out.println("etiquette:");
                    char etiquette = scanner.next().charAt(0);
                    scanner.nextLine(); // Consume newline
                    a.addTransition(srcState, etiquette, dstState);
                } else {
                    ajouterTransitions = false;
                }
            }

            System.out.println("------------------------------------------------------------");

            a.affichageProcDirecte();

            System.out.println("Entrez la chaîne à analyser:");
            String inputString = scanner.nextLine();

            // Effectuer l'analyse lexicale avec la chaîne fournie par l'utilisateur
            a.analyserLexical(inputString);

            System.out.println("Voulez-vous effectuer une nouvelle analyse sur le même automate? oui/non");
            continuer = "oui".equalsIgnoreCase(scanner.nextLine());
        }

        scanner.close();
    }
```

Exemple d'exécution :






Destination:
1
etiquette:
a
error: la transition(1,a,...) existe déjà.
Voulez-vous ajouter une transition? oui/non
oui
etat source:
1
Destination:
1
etiquette:
b
Voulez-vous ajouter une transition? oui/non
oui
etat source:
1
Destination:
3
etiquette:
c
Voulez-vous ajouter une transition? oui/non
non
----
void etat 1() {
Build completed successfully in 6 sec, 681 ms (4 minutes ago)


non
----
void etat 1() {
char c;
c = getchar();
switch(c) {
case 'a': etat 2(); break;
case 'b': etat 1(); break;
case 'c': etat 3(); break;
default: Erreur();
}
}
void etat 2() {
char c;
c = getchar();
switch(c) {
default: Erreur();
}
}
void etat 3() {
char c;
c = getchar();
switch(c) {
default: Erreur();
}
}
Entrez la chaîne à analyser:

```
Entrez la chaîne à analyser:
aac
Erreur: Transition invalide pour le symbole 'a' depuis l'état '2'.
Voulez-vous effectuer une nouvelle analyse sur le même automate? oui/non
oui
Voulez-vous ajouter un etat? oui/non
non
Voulez-vous ajouter une transition? oui/non
non
------------------------------------------------------------
void etat 1() {
    char c;
    c = getchar();
    switch(c) {
        case 'a': etat 2(); break;
        case 'b': etat 1(); break;
        case 'c': etat 3(); break;
        default: Erreur();
    }
}

void etat 2() {
    char c;
    c = getchar();
    switch(c) {
        default: Erreur();
    }
}
```

```
void etat 3() {
    char c;
    c = getchar();
    switch(c) {
        default: Erreur();
    }
}

Entrez la chaîne à analyser:
bbc
Analyse lexicale réussie.
Voulez-vous effectuer une nouvelle analyse sur le même automate? oui/non
non

Process finished with exit code 0
```

Conclusion

En guise de conclusion, la réalisation de ce projet de compilation en Java a été une expérience enrichissante, nous permettant de passer de la théorie à la pratique. La mise en œuvre d'un automate à états finis, avec sa représentation procédurale directe, a été un défi passionnant qui a consolidé notre compréhension des concepts fondamentaux de la compilation. La modularité du code, la gestion des états et des transitions, ainsi que l'interaction utilisateur, ont été autant d'aspects techniques que nous avons maîtrisés. Ce projet a non seulement affiné nos compétences en programmation Java, mais a également approfondi notre compréhension des automates, ouvrant ainsi la voie à des applications plus avancées dans le domaine de la compilation.