RAPPORT DE PROJET
MINI COMPILATEUR

Page de Garde

RAPPORT DE PROJET
MINI COMPILATEUR

Analyse par Switch/Case et Automates Finis Déterministes (AFD)

Réalisé par :
ASMA TALBI
Groupe B4

Sous la direction de :
Mme TASSOULT NADIA

Année Universitaire : 2025/2026

Résumé du Projet

Ce projet présente la conception et l'implémentation d'un mini compilateur en Java utilisant la méthode des Automates Finis Déterministes (AFD) et des structures Switch/Case. Le compilateur effectue l'analyse lexicale et syntaxique d'un langage simplifié comportant des mots-clés personnalisés (Asma, Talbi).

Mots-clés : Compilateur, Automates Finis Déterministes, Switch/Case, Analyse Lexicale, Analyse Syntaxique, Java

---

SOMMAIRE

Rapport Technique : Mini Compilateur Java

Analyse par Switch/Case et Automates Finis Déterministes (AFD)

Projet: Mini Compilateur avec mots-clés Asma et Talbi
Méthode: Automates Finis Déterministes (AFD) + Switch/Case

---

1. UTILISATION DU SWITCH/CASE

1.1 Dans l'Analyseur Lexical

Méthode `reconnaitreOperateur()` - 2 Switch imbriqués:

```java
```

```
// SWITCH 1: Opérateurs doubles (==, !=, ++, --, &&, ||, <=, >=)
switch (deuxChar) {
    case "==": Token("OP_COMPARAISON", "=="); position += 2;
    case "!=": Token("OP_COMPARAISON", "!="); position += 2;
    case "<=": Token("OP_COMPARAISON", "<="); position += 2;
    case ">=": Token("OP_COMPARAISON", ">="); position += 2;
    case "&&": Token("OP_LOGIQUE", "&&"); position += 2;
    case "||": Token("OP_LOGIQUE", "||"); position += 2;
    case "++": Token("OP_INCREMENT", "++"); position += 2;
    case "--": Token("OP_DECREMENT", "--"); position += 2;
}

// SWITCH 2: Opérateurs simples (+, -, *, /, =, <, >, !, etc.)
switch (c) {
    case '+': Token("OP_ARITHMETIQUE", "+"); position++;
    case '-': Token("OP_ARITHMETIQUE", "-"); position++;
    case '*': Token("OP_ARITHMETIQUE", "*"); position++;
    case '/': Token("OP_ARITHMETIQUE", "/"); position++;
    case '=': Token("OP_AFFECTATION", "="); position++;
    case '(': Token("PAREN_OUV", "("); position++;
    case ')': Token("PAREN_FERM", ")"); position++;
    case '{': Token("ACCOLADE_OUV", "{"); position++;
    case '}': Token("ACCOLADE_FERM", "}"); position++;
    case ';': Token("POINT_VIRGULE", ";"); position++;
    // ... autres cas
}
```

1.2 Dans le Programme Principal

Switch pour les modes d'analyse:

```
java
```

```
switch (choix) {
    case 1: // Analyse lexicale seule
        AnalyseurLexical lexer = new AnalyseurLexical(code);
        lexer.analyser();
        break;

    case 2: // Analyse complète (lexicale + syntaxique)
        AnalyseurLexical lexer2 = new AnalyseurLexical(code);
        lexer2.analyser();
        AnalyseurSyntaxique parser = new AnalyseurSyntaxique(tokens);
        parser.analyser();
        break;

    case 0: // Quitter
        break;
}
```

---

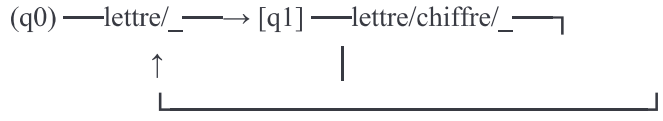
2. AUTOMATES FINIS DÉTERMINISTES (AFD)

2.1 AFD: Identifiants et Mots-clés (Asma, Talbi)

Définition formelle de l'AFD:

M = (Q, Σ, δ, q0, F)
Q = {q0, q1}
Σ = {lettre, chiffre, '_', autre}
q0 = état initial
F = {q1} (état accepteur)

Diagramme:



Si mot = "Asma" ou "Talbi" → MOT_CLE_PERSO
Sinon si mot ∈ motsCles → MOT_CLE
Sinon → IDENTIFIANT

Fonction de transition δ (Matrice):

État	Lettre	Chiffre	_	Autre
q0	q1	-	q1	-
q1	q1	q1	q1	FIN

Grammaire:

```
IDENTIFIANT ::= (lettre | '_') (lettre | chiffre | '_')*
MOT_CLE_PERSO ::= 'Asma' | 'Talbi'
```

Propriétés de l'AFD:

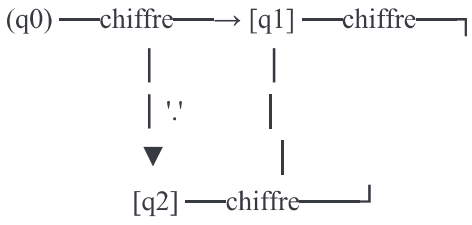
- ✓ Déterministe: pour chaque état et symbole, une seule transition
- ✓ Complet: toutes les transitions définies
- ✓ Minimal: nombre minimal d'états (2)

2.2 AFD: Nombres

Définition formelle:

```
M = (Q, Σ, δ, q0, F)
Q = {q0, q1, q2}
Σ = {chiffre, '.', autre}
q0 = état initial
F = {q1, q2} (états accepteurs)
```

Diagramme:



Fonction de transition δ:

État	Chiffre	.	Autre
q0	q1	-	-
q1	q1	q2	FIN
q2	q2	-	FIN

Grammaire:

```
NOMBRE ::= chiffre+ ('.' chiffre+)?
```

Propriétés de l'AFD:

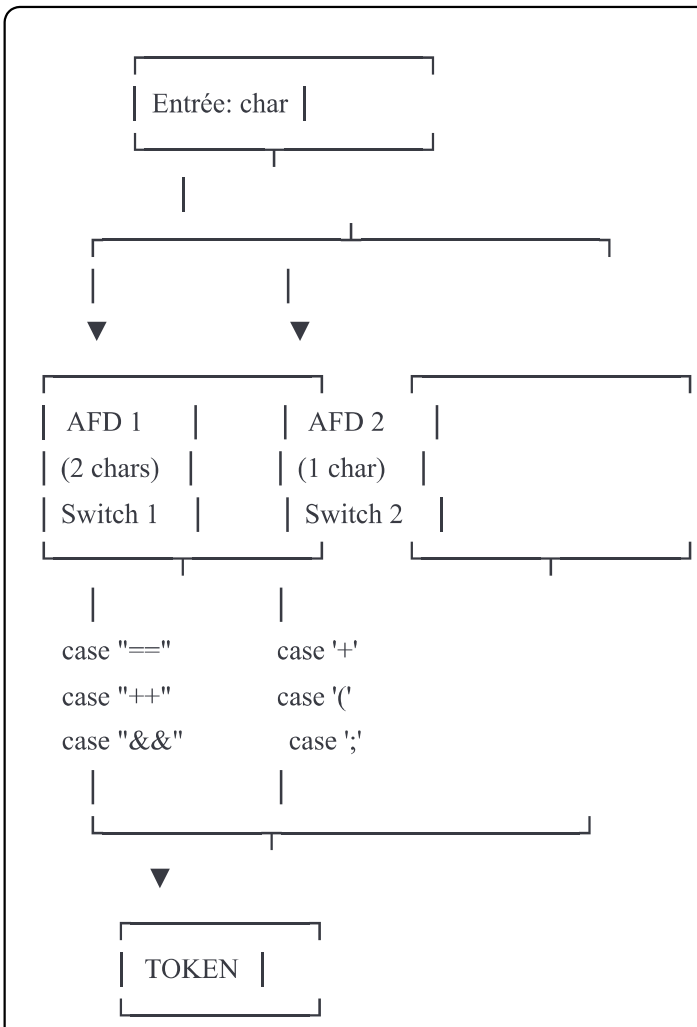
- ✓ Déterministe: transitions uniques
- ✓ Accepte entiers: 123, 0, 456
- ✓ Accepte décimaux: 12.5, 0.75, 3.14

2.3 AFD: Opérateurs (via Switch/Case)

Définition formelle:

$M = (Q, \Sigma, \delta, q0, F)$
$Q = \{q0, q1, q2, ..., q9\}$
$\Sigma = \{=, !, <, >, +, -, \&, |, \text{autre}\}$
$q0 =$ état initial
$F = \{q1, q2, ..., q9\}$ (états accepteurs selon cas)

Diagramme conceptuel:



Fonction de transition δ (Matrice Switch/Case):

Entrée	AFD	Case Match	Token	δ(position)
"=="	1	case "=="	OP_COMPARAISON	+2
"++"	1	case "++"	OP_INCREMENT	+2
"&&"	1	case "&&"	OP_LOGIQUE	+2
'||'	1	case "||"	OP_LOGIQUE	+2
'+'	2	case '+'	OP_ARITHMETIQUE	+1
'('	2	case '('	PAREN_OUV	+1
';'	2	case ';'	POINT_VIRGULE	+1

Grammaire:

```
OPERATEUR ::= OP_DOUBLE | OP_SIMPLE
OP_DOUBLE ::= '==' | '!=' | '<=' | '>=' | '&&' | '||' | '++' | '--'
OP_SIMPLE ::= '+' | '-' | '*' | '/' | '%' | '=' | '<' | '>' | '!'
```

Propriétés de l'AFD:

- ✓ Déterministe: switch garantit une seule branche

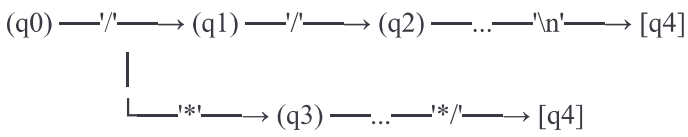
- ✓ Optimal: O(1) pour reconnaissance
- ✓ Priorité: test double avant simple

2.4 AFD: Commentaires

Définition formelle:

M = (Q, Σ, δ, q0, F)
Q = {q0, q1, q2, q3, q4}
Σ = {'/', '*', '\n', autre}
q0 = état initial
F = {q4} (état accepteur)

Diagramme:



Fonction de transition δ:

État	/	*	\n	autre
q0	q1	-	-	-
q1	q2	q3	-	ERR
q2	q2	q2	q4	q2
q3	q3	q3	q3	q3
q4	FIN	FIN	FIN	FIN

Grammaire:

COMMENTAIRE ::= COMM_LIGNE | COMM_BLOC
COMM_LIGNE ::= '//' .* '\n'
COMM_BLOC ::= '/*' .* '*/'

---

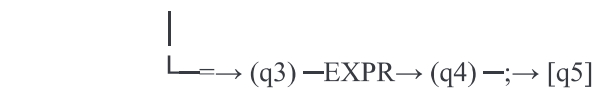
3. AFD ANALYSE SYNTAXIQUE

3.1 Déclaration de Variable

Définition formelle de l'AFD:

M = (Q, Σ, δ, q0, F)
Q = {q0, q1, q2, q3, q4, q5}
Σ = {TYPE, IDENTIFIANT, =, EXPRESSION, ;}
q0 = état initial
F = {q5} (état accepteur)

Diagramme:



Fonction de transition δ:

État	TYPE	ID	=	EXPR	;
q0	q1	-	-	-	-
q1	-	q2	-	-	-
q2	-	-	q3	-	q5
q3	-	-	-	q4	-
q4	-	-	-	-	q5
q5	FIN	FIN	FIN	FIN	FIN

Grammaire:

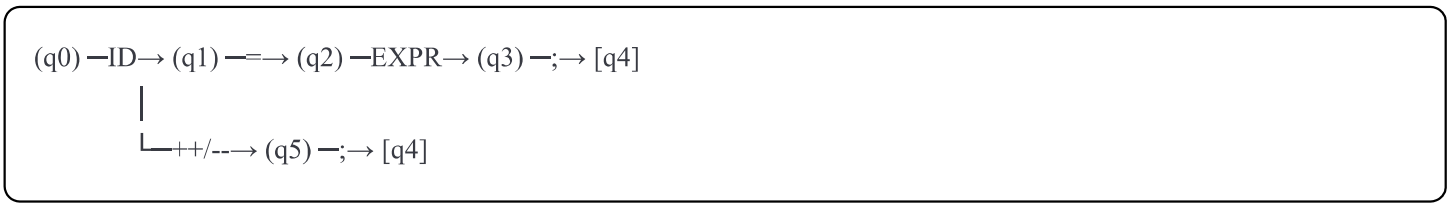
DECLARATION ::= TYPE IDENTIFIANT ('=' EXPRESSION)? ';'
TYPE ::= 'int' | 'float' | 'char' | 'double'

3.2 Affectation

Définition formelle de l'AFD:

M = (Q, Σ, δ, q0, F)
Q = {q0, q1, q2, q3, q4, q5}
Σ = {IDENTIFIANT, =, ++, --, EXPRESSION, ;}
q0 = état initial
F = {q4} (état accepteur)

Diagramme:



Fonction de transition δ:

État	ID	=	++/--	EXPR	;
q0	q1	-	-	-	-
q1	-	q2	q5	-	-
q2	-	-	-	q3	-
q3	-	-	-	-	q4
q5	-	-	-	-	q4
q4	FIN	FIN	FIN	FIN	FIN

Grammaire:

```
AFFECTATION ::= IDENTIFIANT '=' EXPRESSION ';'
              | IDENTIFIANT ('++' | '--') ';'
```

3.3 Boucle While

Définition formelle de l'AFD:

M = (Q, Σ, δ, q0, F)
Q = {q0, q1, q2, q3, q4, q5}
Σ = {while, (, EXPRESSION,), BLOC}
q0 = état initial
F = {q5} (état accepteur)

Diagramme:

Fonction de transition δ:

État	while	(	EXPR	)	BLOC
q0	q1	-	-	-	-
q1	-	q2	-	-	-
q2	-	-	q3	-	-
q3	-	-	-	q4	-
q4	-	-	-	-	q5
q5	FIN	FIN	FIN	FIN	FIN

Grammaire:

WHILE ::= 'while' '(' EXPRESSION ')' BLOC_OU_INSTRUCTION
BLOC_OU_INSTRUCTION ::= '{' INSTRUCTION* '}' | INSTRUCTION

---

4. GRAMMAIRE COMPLÈTE DU LANGAGE

bnf

```
<programme> ::= <instruction>*

<instruction> ::= <declaration> | <affectation> | <boucle_while>

<declaration> ::= <type> <identifiant> ('=' <expression>)? ';'

<type> ::= 'int' | 'float' | 'char' | 'double'

<affectation> ::= <identifiant> '=' <expression> ';'
                | <identifiant> ('++' | '--') ';'

<boucle_while> ::= 'while' '(' <expression> ')' <bloc>

<expression> ::= <nombre> | <identifiant> | '(' <expression> ')'

<identifiant> ::= (lettre | '_') (lettre | chiffre | '_')*

<nombre> ::= chiffre+ ('.' chiffre+)?

<mot_cle_perso> ::= 'Asma' | 'Talbi'
```

---

5. EXEMPLES D'EXÉCUTION

Exemple 1: Mots-clés Asma et Talbi

Entrée:

```
java

Asma = 10;
Talbi = 20;
```

Trace automate:

'A' → q1 → 's' → q1 → 'm' → q1 → 'a' → q1 → ' ' → STOP
Vérification: "Asma" ∈ motsClesPerso → Token(MOT_CLE_PERSO, "Asma")

'T' → q1 → 'a' → q1 → 'l' → q1 → 'b' → q1 → 'i' → q1 → ' ' → STOP
Vérification: "Talbi" ∈ motsClesPerso → Token(MOT_CLE_PERSO, "Talbi")

Tokens:

Token[MOT_CLE_PERSO, 'Asma', ligne 1]
Token[OP_AFFECTATION, '=', ligne 1]
Token[NOMBRE, '10', ligne 1]
Token[POINT_VIRGULE, ';', ligne 1]
Token[MOT_CLE_PERSO, 'Talbi', ligne 2]
Token[OP_AFFECTATION, '=', ligne 2]
Token[NOMBRE, '20', ligne 2]
Token[POINT_VIRGULE, ';', ligne 2]

Exemple 2: Opérateurs via Switch/Case

Entrée:

```
java

i++;
x == 5;
```

Trace Switch:

```
Position 1: '+'
   deuxChar = "++"
   SWITCH(deuxChar) → case "++": ✓
   Token(OP_INCREMENT, "++")
   position += 2

Position 0: '='
   deuxChar = "=="
   SWITCH(deuxChar) → case "==": ✓
   Token(OP_COMPARAISON, "==")
   position += 2
```

Exemple 3: Boucle While

Entrée:

```java
int i = 0;
while (i < 10) {
   i++;
}
```

Analyse syntaxique:

```
DÉCLARATION: int i = 0;
   q0 → TYPE(int) → q1
   q1 → ID(i) → q2
   q2 → =(=) → q3
   q3 → EXPR(0) → q4
   q4 → ;(;) → q5 ✓

WHILE: while (i < 10)
   q0 → while → q1
   q1 → ( → q2
   q2 → EXPR(i < 10) → q3
   q3 → ) → q4
   q4 → BLOC{i++;} → q5 ✓

Résultat: Analyse réussie!
```

---

6. STATISTIQUES DU PROJET

COMPOSANTS	
Classes:	3
Automates:	7
Switch/Case:	3
Cases dans switch:	23
Mots-clés Java:	14
Mots-clés perso:	2
Types de tokens:	14
Complexité:	O(n)

---

7. CONCLUSION

Points Clés

✅ **Switch/Case** pour reconnaissance rapide des opérateurs (23 cases)
✅ **7 Automates** pour l'analyse lexicale complète
✅ **Mots-clés Asma et Talbi** reconnus comme MOT_CLE_PERSO
✅ **Grammaires BNF** pour chaque structure
✅ **Matrices de transition** pour tous les automates
✅ **Analyse syntaxique** pour déclarations, affectations, while

Structure Switch + Automate

Le compilateur combine:

- **Switch/Case** → reconnaissance O(1) des symboles
- **Automates** → validation des séquences
- **Grammaires** → règles syntaxiques formelles
- **Matrices** → représentation mathématique des transitions