Université Abderrahmane Mira – Béjaïa
Faculté des Sciences Exactes – Département d'Informatique

Projet : Analyseurs Lexicale et Syntaxique d'opérateurs ternaires en java

Réalisé par : **Boufadene hanane**
Encadré par : **Mlle N.Tassoult**
Groupe : **A3**

Date : 8 décembre 2025

Table des matières

1 Introduction

Dans ce projet, nous avons développé deux composants essentiels du processus de compilation :

— un **analyseur lexical**, chargé de découper une expression en unités lexicales (tokens) ;

— un **analyseur syntaxique**, chargé de vérifier que l'expression respecte une grammaire précise.

Le programme vise à analyser des expressions de la forme :

condition ? valeur_vraie : valeur_fausse ;

Il s'agit du principe de l'opérateur ternaire utilisé dans de nombreux langages.

L'analyse syntaxique repose sur un parcours caractère par caractère, tandis que l'analyse lexicale repose sur un **automate déterministe fini (ADF)** représenté par une matrice de transitions.

2 Analyseur Lexical

2.1 Rôle de l'analyseur lexical

L'analyseur lexical transforme la chaîne d'entrée en une séquence de tokens. Chaque token représente une unité significative : identifiant, nombre, opérateur, parenthèse, point-virgule, etc.

Il se compose :

— d'un automate fini déterministe (ADF) ;
— d'une fonction de catégorisation des caractères ;
— d'une génération des tokens à partir des états atteints.

2.2 Catégories de caractères

— lettres ou underscore (`a-z, A-Z, _`) ;
— chiffres (`0-9`) ;
— opérateurs relationnels (`< > = !`) ;
— point d'interrogation (`?`) ;
— deux-points (`:`) ;
— parenthèses (`( )`) ;
— point-virgule (`;`) ;
— espaces ;
— caractère invalide.

2.3 Automate déterministe

Matrice[*état*][*catégorie*]

État	Signification
0	état initial
1	identifiant
2	nombre
3	opérateur composé (`<>, <=, >=, !=`)
4	opérateur `?`
5	opérateur `:`
6	parenthèse ouvrante
7	parenthèse fermante
8	état d'erreur (caractère invalide)
9	point-virgule

2.4 Fonctionnement

L'analyseur parcourt la chaîne caractère par caractère, détermine la catégorie du caractère, puis utilise la matrice de transitions pour passer d'un état à l'autre. Lorsqu'un état final est atteint, un token correspondant est créé et ajouté à la liste des résultats. À la fin de l'analyse, un token de fin (#) est ajouté pour marquer la fin de la séquence.

2.5 Code source complet

```
package a3hananeboufadene;

import java.util.*;

public class AnalyseurLexical {

    public enum TokenType {
        IDENTIFIANT, NOMBRE, OPERATEUR_QUESTION,
           OPERATEUR_DEUX_POINTS,
        OPERATEUR_COMP, PARENTHESE_OUV, PARENTHESE_FERM,
           POINT_VIRGULE,
        FIN, INVALIDE
    }

    public static class Token {
        TokenType type;
        String valeur;

        public Token(TokenType type, String valeur) {
            this.type = type;
            this.valeur = valeur;
        }

        @Override
        public String toString() {
            return "<" + type + ", \"" + valeur + "\">";
        }
    }

    private static final int[][] MATRICE = {
            {1, 2, 3, 4, 5, 6, 7, 0, 8, 9},
            {1, 1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, 2, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, 3, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
            {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
    };

    private final String texte;
    private int index;
    private final List<Token> resultat;

    public AnalyseurLexical(String texte) {
        this.texte = texte;
        this.index = 0;
```

```
48          this.resultat = new ArrayList<>();
49      }
50
51      private int getCategorie(char c) {
52          if (Character.isLetter(c) || c == '_') return 0;
53          if (Character.isDigit(c)) return 1;
54          if ("<>!= ".indexOf(c) >= 0 && c != ' ') return 2;
55          if (c == '?') return 3;
56          if (c == ':') return 4;
57          if (c == '(') return 5;
58          if (c == ')') return 6;
59          if (Character.isWhitespace(c)) return 7;
60          if (c == ';') return 9;
61          return 8;
62      }
63
64      public List<Token> analyser() {
65          while (index < texte.length()) {
66              int etat = 0;
67              StringBuilder lexeme = new StringBuilder();
68              int depart = index;
69
70              while (index < texte.length()) {
71                  char c = texte.charAt(index);
72                  int categorie = getCategorie(c);
73                  int suivant = MATRICE[etat][categorie];
74
75                  if (suivant == -1) break;
76                  if (suivant != 0) lexeme.append(c);
77                  etat = suivant;
78                  index++;
79                  if (etat == 4 || etat == 5 || etat == 6 || etat
                        == 7 || etat == 9) break;
80              }
81
82              if (lexeme.length() > 0) {
83                  Token t = creerToken(etat, lexeme.toString());
84                  if (t != null) resultat.add(t);
85              } else if (etat == 0 && depart == index) {
86                  index++;
87              }
88          }
89
90          resultat.add(new Token(TokenType.FIN, "#"));
91          return resultat;
92      }
93
94      private Token creerToken(int etat, String lexeme) {
95          switch (etat) {
96              case 1: return new Token(TokenType.IDENTIFIANT,
                    lexeme);
```

```
97              case 2: return new Token(TokenType.NOMBRE, lexeme);
98              case 3: return new Token(TokenType.OPERATEUR_COMP,
                    lexeme);
99              case 4: return new Token(TokenType.OPERATEUR_QUESTION
                    , lexeme);
100             case 5: return new Token(TokenType.
                    OPERATEUR_DEUX_POINTS, lexeme);
101             case 6: return new Token(TokenType.PARENTHESE_OUV,
                    lexeme);
102             case 7: return new Token(TokenType.PARENTHESE_FERM,
                    lexeme);
103             case 9: return new Token(TokenType.POINT_VIRGULE,
                    lexeme);
104             case 8: return new Token(TokenType.INVALIDE, lexeme);
105         }
106         return null;
107     }
108
109     public void afficherTokens() {
110         resultat.forEach(System.out::println);
111     }
112 }
```

3 Analyseur Syntaxique

3.1 Rôle de l'analyseur syntaxique

L'analyseur syntaxique vérifie que l'expression respecte la grammaire de l'opérateur ternaire avec un point-virgule à la fin :

$$E \rightarrow C \; ? \; V \; : \; V \; ;$$

avec les sous-règles :

$$\begin{aligned} C &\rightarrow T \mid T \text{ OP } T \\ T &\rightarrow \text{IDENT} \mid \text{NOMBRE} \mid (E) \\ V &\rightarrow T \\ \text{OP} &\rightarrow == \mid != \mid < \mid > \mid <= \mid >= \end{aligned}$$

3.2 Description du fonctionnement

Le parseur suit ces étapes :

1. Lecture de la première opérande (T) ;
2. Lecture éventuelle d'un opérateur relationnel ;
3. Lecture de la deuxième opérande si un opérateur a été trouvé ;
4. Vérification du symbole `?` ;
5. Lecture de la partie vraie (identifiant, nombre ou parenthèses) ;
6. Vérification du symbole `:` ;
7. Lecture de la partie fausse ;
8. Vérification du point-virgule `;` ;
9. Vérification que la chaîne est complètement consommée.

3.3 Gestion des erreurs

L'expression est incorrecte si :
— parenthèses non fermées ;
— opérateur relationnel incomplet ;
— absence du point d'interrogation `?` ;
— absence du deux-points `:` ;
— absence du point-virgule `;` à la fin ;
— caractère non reconnu ;
— expression restante après analyse.

3.4 Code source complet

```
1 package a3hananeboufadene;
2
3 import java.util.Scanner;
4
5 public class AnalyseurSyntaxique {
```

```
6
7       public static int lireOperande(String expr, int pos,
            StringBuilder erreur) {
8           int n = expr.length();
9           if (pos >= n) {
10              erreur.append("Operande attendu mais trouv  fin de l
                    'expression.");
11              return -1;
12          }
13          char c = expr.charAt(pos);
14          if (Character.isLetterOrDigit(c)) {
15              while (pos < n && (Character.isLetterOrDigit(expr.
                    charAt(pos)) || expr.charAt(pos) == '_')) pos++;
16              return pos;
17          }
18          if (c == '(') {
19              pos++;
20              while (pos < n && expr.charAt(pos) != ')') pos++;
21              if (pos < n && expr.charAt(pos) == ')') return pos +
                    1;
22              else { erreur.append("Parenth se '(' sans ')'
                    correspondante."); return -1; }
23          }
24          erreur.append("Operande invalide    la position ").append
                (pos).append(".");
25          return -1;
26      }
27
28      public static boolean analyserExpression(String expr,
            StringBuilder erreur) {
29          expr = expr.replace(" ", "");
30          int i = 0, n = expr.length();
31          boolean correcte = true;
32
33          int next = lireOperande(expr, i, erreur);
34          if (next == -1) correcte = false; else i = next;
35
36          boolean comparaisonTrouvee = false;
37          if (correcte && i < n) {
38              if (expr.startsWith("==", i) || expr.startsWith("!=",
                    i)
39                      || expr.startsWith("<=", i) || expr.
                            startsWith(">=", i)) {
40                  comparaisonTrouvee = true;
41                  i += 2;
42              } else if (expr.charAt(i) == '<' || expr.charAt(i) ==
                    '>') {
43                  comparaisonTrouvee = true;
44                  i++;
45              }
46          }
```

```
47
48          if (correcte && comparaisonTrouvee) {
49              int old = i;
50              next = lireOperande(expr, i, erreur);
51              if (next == -1) {
52                  correcte = false;
53                  if (erreur.length() == 0) erreur.append("Operande
                        droite invalide    la position ").append(old).
                        append(".");
54              } else i = next;
55          }
56
57          if (correcte) {
58              if (i < n && expr.charAt(i) == '?') i++;
59              else { correcte = false; erreur.append("Symbole '?'
                    attendu    la position ").append(i).append("."); }
60          }
61
62          if (correcte) {
63              int old = i;
64              next = lireOperande(expr, i, erreur);
65              if (next == -1) {
66                  correcte = false;
67                  if (erreur.length() == 0) erreur.append("
                        Expression apr s '?' invalide    la position "
                        ).append(old).append(".");
68              } else i = next;
69          }
70
71          if (correcte) {
72              if (i < n && expr.charAt(i) == ':') i++;
73              else { correcte = false; erreur.append("Symbole ':'
                    attendu    la position ").append(i).append("."); }
74          }
75
76          if (correcte) {
77              int old = i;
78              next = lireOperande(expr, i, erreur);
79              if (next == -1) {
80                  correcte = false;
81                  if (erreur.length() == 0) erreur.append("
                        Expression apr s ':' invalide    la position "
                        ).append(old).append(".");
82              } else i = next;
83          }
84
85          if (correcte) {
86              if (i < n && expr.charAt(i) == ';') i++;
87              else { correcte = false; erreur.append("Point-virgule
                      ';' attendu    la fin de l'expression    la
                    position ").append(i).append("."); }
```

```
        }

        if (correcte && i != n) {
            correcte = false;
            erreur.append("Caract res inattendus apr s le point
               -virgule    partir de la position ").append(i).
               append(".");
        }

        return correcte;
    }
}
```

4 Code source de MainAnalyseur.java

```
package a3hananeboufadene;

import a3hananeboufadene.AnalyseurSyntaxique;
import a3hananeboufadene.AnalyseurLexical;
import java.util.Scanner;

public class MainAnalyseur {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String rep = "o";

        while (rep.equals("o") || rep.equals("oui")) {
            System.out.print("ATTENTION!! l'expression doit etre
               de cette maniere x?y:z; \n");
            System.out.print("Entrer une expression : ");
            String expr = sc.nextLine();

            System.out.println("\n--- Analyse Lexicale ---");
            AnalyseurLexical lexical = new AnalyseurLexical(expr)
               ;
            lexical.analyser();
            lexical.afficherTokens();

            System.out.println("\n--- Analyse Syntaxique ---");
            StringBuilder erreur = new StringBuilder();
            boolean correcte = AnalyseurSyntaxique.
               analyserExpression(expr, erreur);
            if (correcte) System.out.println("Expression correcte
                !");
            else System.out.println("Expression INCORRECTE !\
               nErreur : " + erreur);

            System.out.print("\nVoulez-vous analyser une autre
               expression ? (o/n) : ");
```

```
            rep = sc.nextLine().trim().toLowerCase();
            System.out.println();
        }

        System.out.println("Fin du programme.");
        sc.close();
    }
}
```

4.1 Exemple d'analyse correcte

$$(x < 10)?y_1 : 25;$$

Sortie :

```
Expression correcte !
```

4.2 Exemple d'expression incorrecte

$$x?y : z$$

Sortie :

```
Expression INCORRECTE !
Erreur : Point-virgule ';' attendu à la fin de l'expression à la position 7.
```

5 Conclusion

La réalisation de ce projet a permis de mettre en pratique les concepts fondamentaux liés aux premières phases de la compilation. Nous avons tout d'abord conçu un analyseur lexical fondé sur un automate déterministe, capable d'identifier et de classifier différents types de lexèmes tels que les identifiants, les nombres, les opérateurs relationnels et les symboles spécifiques à l'opérateur ternaire. Cette étape a montré l'importance d'un découpage précis et rigoureux des unités lexicales, puisque la qualité de l'analyse syntaxique en dépend directement.

L'analyseur syntaxique, développé en complément, a permis de valider la structure grammaticale de l'expression selon les règles d'une grammaire bien définie. Son fonctionnement, basé sur un parcours séquentiel et des contrôles successifs, a mis en évidence la nécessité d'une gestion méthodique des structures imbriquées, telles que les parenthèses, ainsi que la cohérence de l'expression ternaire dans son ensemble.