
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 ;
— chiffres ;
— 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 Code source complet

```
1  package com.mycompany.projetcompil;
2
3  import java.util.*;
4
5  public class AnalyseurLexical {
6
7      enum TokenType {
```

```
        IDENTIFIANT, NOMBRE, OPERATEUR_QUESTION,
           OPERATEUR_DEUX_POINTS,
        OPERATEUR_COMP, PARENTHESE_OUV, PARENTHESE_FERM,
           POINT_VIRGULE,
        FIN, INVALIDE
    }

    static class Token {
        TokenType type;
        String valeur;
        Token(TokenType type, String valeur) { this.type = type;
           this.valeur = valeur; }
        @Override
        public String toString() { return "<" + type + ", \"" +
           valeur + "\">"; }
    }

    private static final int[][] MATRICE = {
        /*0*/ {1,2,3,4,5,6,7,0,8,9},
        /*1*/ {1,1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*2*/ {-1,2,-1,-1,-1,-1,-1,-1,-1,-1},
        /*3*/ {-1,-1,3,-1,-1,-1,-1,-1,-1,-1},
        /*4*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*5*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*6*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*7*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*8*/ {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
        /*9*/ {-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; this.resultat = new ArrayList<>(); }

    private int getCategorie(char c) {
        if (Character.isLetter(c) || c=='_') return 0;
        if (Character.isDigit(c)) return 1;
        if ("<>!= ".indexOf(c)>=0 && c!=' ') return 2;
        if (c=='?') return 3;
        if (c==':') return 4;
        if (c=='(') return 5;
        if (c==')') return 6;
        if (Character.isWhitespace(c)) return 7;
        if (c==';') return 9;
        return 8;
    }

    public List<Token> analyser() {
```

```
54          while (index<texte.length()) {
55              int etat=0;
56              StringBuilder lexeme=new StringBuilder();
57              int depart=index;
58              while(index<texte.length()) {
59                  char c=texte.charAt(index);
60                  int categorie=getCategorie(c);
61                  int suivant=MATRICE[etat][categorie];
62                  if(suivant==-1) break;
63                  if(suivant!=0) lexeme.append(c);
64                  etat=suivant;
65                  index++;
66                  if(etat==4||etat==5||etat==6||etat==7||etat==9)
                        break;
67              }
68              if(lexeme.length()>0){
69                  Token t=creerToken(etat,lexeme.toString());
70                  if(t!=null) resultat.add(t);
71              } else if(etat==0 && depart==index) index++;
72          }
73          resultat.add(new Token(TokenType.FIN,"#"));
74          return resultat;
75      }
76
77      private Token creerToken(int etat,String lexeme){
78          switch(etat){
79              case 1: return new Token(TokenType.IDENTIFIANT,lexeme
                    );
80              case 2: return new Token(TokenType.NOMBRE,lexeme);
81              case 3: return new Token(TokenType.OPERATEUR_COMP,
                    lexeme);
82              case 4: return new Token(TokenType.OPERATEUR_QUESTION
                    ,lexeme);
83              case 5: return new Token(TokenType.
                    OPERATEUR_DEUX_POINTS,lexeme);
84              case 6: return new Token(TokenType.PARENTHESE_OUV,
                    lexeme);
85              case 7: return new Token(TokenType.PARENTHESE_FERM,
                    lexeme);
86              case 9: return new Token(TokenType.POINT_VIRGULE,
                    lexeme);
87              case 8: return new Token(TokenType.INVALIDE,lexeme);
88          }
89          return null;
90      }
91
92      public void afficherTokens(){ resultat.forEach(System.out::
            println); }
93
94      public static void main(String[] args){
95          Scanner sc=new Scanner(System.in);
```

```
96          String rep="o";
97          while(rep.equals("o") || rep.equals("oui")){
98              System.out.print("ATTENTION!! l'expression doit etre
                    de cette maniere x?y:z; \n");
99              System.out.print(" Entrer une expression : ");
100             String expr=sc.nextLine();
101             System.out.println("\n--- Analyse de : "+expr+" ---\n
                    ");
102             AnalyseurLexical a=new AnalyseurLexical(expr);
103             a.analyser();
104             a.afficherTokens();
105             System.out.print("\nVoulez-vous analyser une autre
                    expression ? (o/n) : ");
106             rep=sc.nextLine().trim().toLowerCase();
107             System.out.println();
108         }
109         sc.close();
110     }
111 }
```

# 3 Analyseur Syntaxique avec point-virgule

## 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 com.mycompany.projetcompil;
2
3 import java.util.Scanner;
4
5 public class AnalyseurSyntaxique01 {
```

```
6
7       private 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 {
23                  erreur.append("Parenth se '(' sans ')'
                        correspondante.");
24                  return -1;
25              }
26          }
27          erreur.append("Operande invalide    la position ").append
                (pos).append(".");
28          return -1;
29      }
30
31      public static void main(String[] args) {
32          Scanner sc = new Scanner(System.in);
33          boolean continuer = true;
34
35          while (continuer) {
36              System.out.print("Entrez une expression : ");
37              String expr = sc.nextLine().replace(" ", "");
38              int i = 0;
39              int n = expr.length();
40              boolean correcte = true;
41              StringBuilder erreur = new StringBuilder();
42
43              int next = lireOperande(expr, i, erreur);
44              if (next == -1) correcte = false; else i = next;
45
46              boolean comparaisonTrouvee = false;
47              if (correcte && i < n) {
48                  if (expr.startsWith("==", i) || expr.startsWith("
                        !=", i)
```

```
49                      || expr.startsWith("<=", i) || expr.
                           startsWith(">=", i)) {
50                      comparaisonTrouvee = true;
51                      i += 2;
52                  } else if (expr.charAt(i) == '<' || expr.charAt(i
                       ) == '>') {
53                      comparaisonTrouvee = true;
54                      i++;
55                  }
56              }
57
58              if (correcte && comparaisonTrouvee) {
59                  int old = i;
60                  next = lireOperande(expr, i, erreur);
61                  if (next == -1) {
62                      correcte = false;
63                      if (erreur.length() == 0) erreur.append("
                           Operande droite invalide    la position ").
                           append(old).append(".");
64                  } else i = next;
65              }
66
67              if (correcte) {
68                  if (i < n && expr.charAt(i) == '?') i++;
69                  else { correcte = false; erreur.append("Symbole
                       '?' attendu    la position ").append(i).append(
                       "."); }
70              }
71
72              if (correcte) {
73                  int old = i;
74                  next = lireOperande(expr, i, erreur);
75                  if (next == -1) {
76                      correcte = false;
77                      if (erreur.length() == 0) erreur.append("
                           Expression apr s '?' invalide    la
                           position ").append(old).append(".");
78                  } else i = next;
79              }
80
81              if (correcte) {
82                  if (i < n && expr.charAt(i) == ':') i++;
83                  else { correcte = false; erreur.append("Symbole
                       ':' attendu    la position ").append(i).append(
                       "."); }
84              }
85
86              if (correcte) {
87                  int old = i;
88                  next = lireOperande(expr, i, erreur);
89                  if (next == -1) {
```

```
                    correcte = false;
                    if (erreur.length() == 0) erreur.append("
                        Expression apr s ':' invalide    la
                        position ").append(old).append(".");
                } else i = next;
            }

            if (correcte) {
                if (i < n && expr.charAt(i) == ';') i++;
                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(".");
            }

            if (correcte) System.out.println("Expression correcte
                !");
            else { System.out.println("Expression INCORRECTE !");
                System.out.println("Erreur : " + erreur); }

            System.out.print("Voulez-vous entrer une autre
                expression ? (o/n) : ");
            if (sc.nextLine().equalsIgnoreCase("n")) continuer =
                false;
        }

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

## 3.5 Exemple d'analyse correcte

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

Sortie :

```
Expression correcte !
```

## 3.6 Exemple d'expression incorrecte

$$x?y : z$$

Sortie :

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

# 4 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.