Projet Compilation Analyseur Lexical En Java

Réalisé par : **Soufiane ATTIF**
2 G I

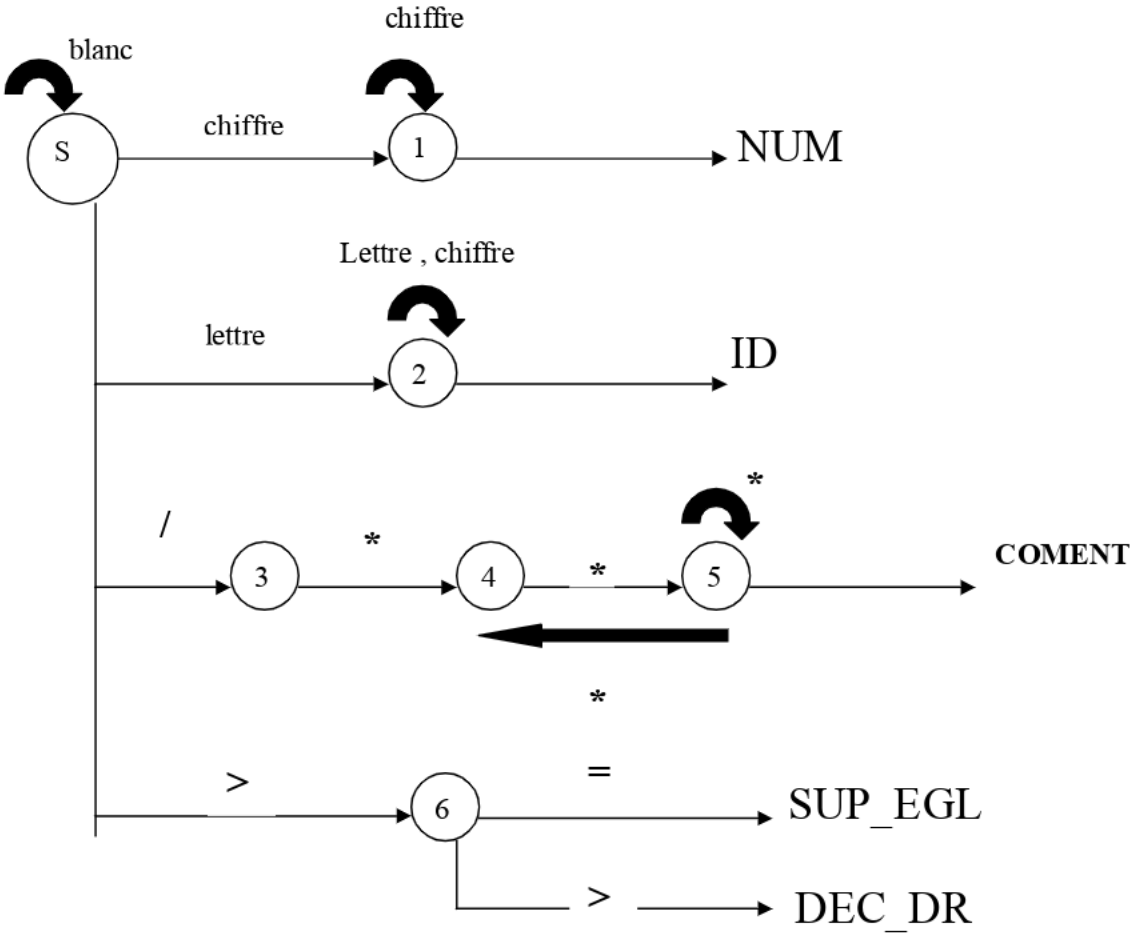
I. Présentation de Projet :

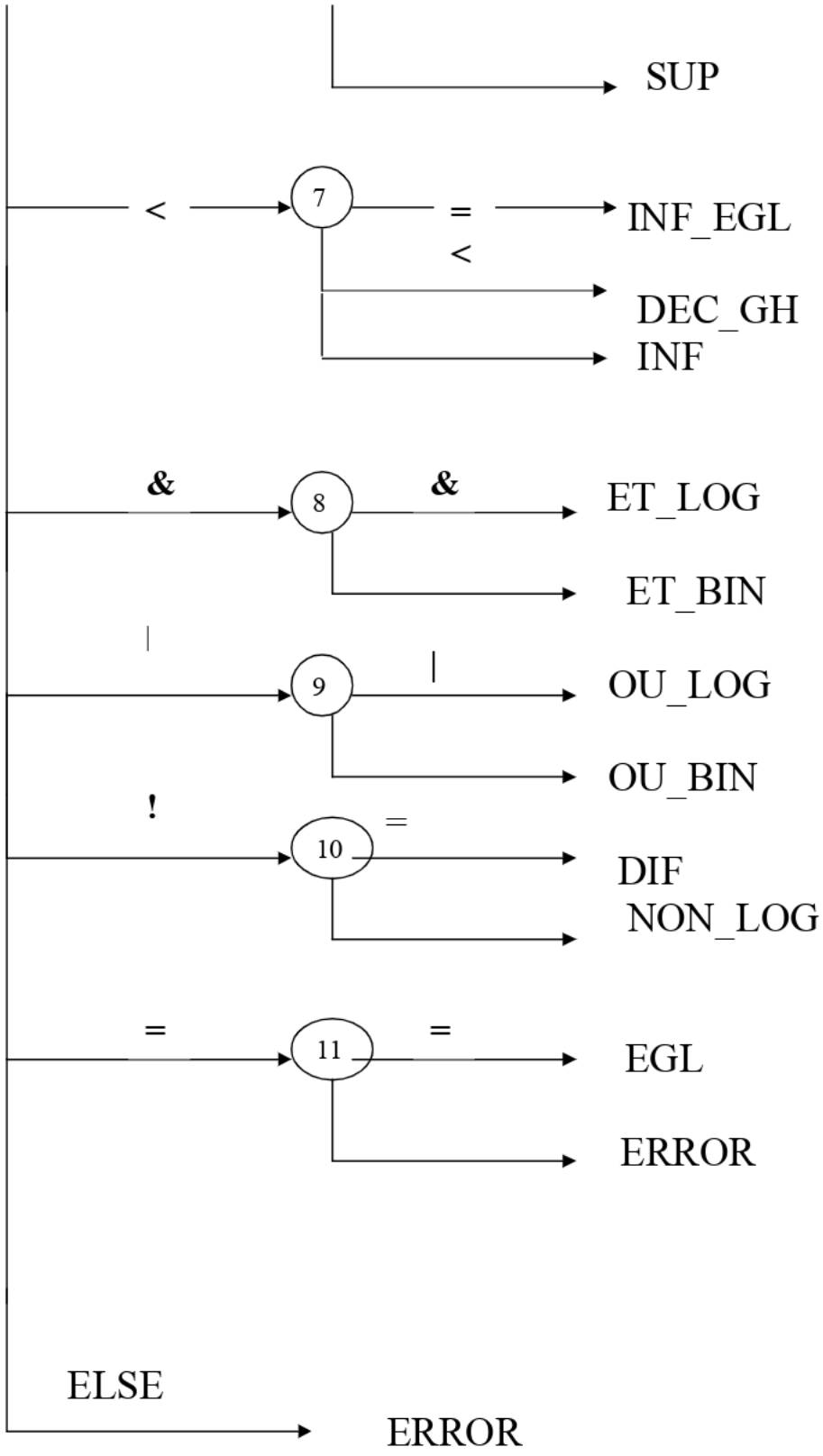
Le but est de réaliser un analyseur syntaxique d'un sous ensemble du langage C appelé C -- .
Le langage est formé principalement par :

- Des mots clés : if, then, else, extern, for, void, while, return et int.
- Des Identificateurs : commençant par une lettre et formés de lettres et chiffres.
- Des opérateurs : <, >, <<,>>,+,- ,*, /,&,&&,|,| |, !, !=,==,<=,>=.
- Des commentaires : /* blabla. …..*/

II. Diagramme d'état :

Ce langage est régit par l'automate suivant :




SUP
<
7
=
INF_EGL
<
DEC_GH
INF
&
8
&
ET_LOG
ET_BIN
|
9
|
OU_LOG
OU_BIN
!
10
=
DIF
NON_LOG
=
11
=
EGL
ERROR
ELSE
ERROR

III.Programme :

Ce projet réalisé en java, est divisé en 3 classes :

- ↘ Token : classes où sont définis les tokens du langages.
- ↘ Scanner : classes où définit le traitement du flux du fichier de lecture.
- ↘ scannerTest : classe permettant de boucler sur les traitement.
- ↘ Les classes graphiques.

Ces classes forment un package nommé compilateur, et font appel l'une à l'autre suivant des principes de traitement du flux.
Le programme lit un fichier à partir d'un emplacement fournit et détermine les lexèmes et les erreurs commises.

Le code :

Compilateur.java

```
package compilateur;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.io.*;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.*;
import javax.swing.JTextField;
import javax.swing.JToggleButton;
import javax.swing.JButton;
import javax.swing.JTextArea;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JScrollPane;


public class compilateur extends JFrame {
   JPanel contentPane;
   JLabel jLabel1 = new JLabel();
   JLabel jLabel2 = new JLabel();
   JTextField jTextField1 = new JTextField();
```

```
JButton jButton1 = new JButton();
JScrollPane jScrollPane1 = new JScrollPane();
JTextArea jTextArea1 = new JTextArea();
JLabel jLabel3 = new JLabel();
JScrollPane jScrollPane2 = new JScrollPane();
JTextArea jTextArea2 = new JTextArea();
public compilateur() {
   try {
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      jbInit();
   } catch (Exception exception) {
      exception.printStackTrace();
   }
}

private void jbInit() throws Exception {
   contentPane = (JPanel) getContentPane();
   contentPane.setLayout(null);
   setSize(new Dimension(469, 500));
   setTitle("Analyseur lexical");
   jLabel1.setFont(new java.awt.Font("Garamond", Font.BOLD, 20));
   jLabel1.setText("Analyseur Lexical");
   jLabel1.setBounds(new Rectangle(124, 4, 159, 39));
   jLabel2.setText("Chemin :");
   jLabel2.setBounds(new Rectangle(12, 45, 47, 23));
   jTextField1.setBounds(new Rectangle(54, 47, 178, 21));
   jButton1.setBounds(new Rectangle(288, 48, 94, 20));
   jButton1.setText("Compiler");
   jButton1.addActionListener(new compilateur_jButton1_actionAdapter(this));
   jTextField1.addActionListener(new
                                 compilateur_jTextField1_actionAdapter(this));
   jScrollPane1.setBounds(new Rectangle(12, 81, 372, 200));
   jLabel3.setText("Erreurs :");
   jLabel3.setBounds(new Rectangle(13, 289, 49, 23));
   jScrollPane2.setBounds(new Rectangle(12, 310, 375, 100));
   contentPane.add(jLabel1);
   contentPane.add(jLabel2);
   contentPane.add(jTextField1);
   contentPane.add(jButton1);
   contentPane.add(jScrollPane1);
   contentPane.add(jLabel3);
   contentPane.add(jScrollPane2);
   jScrollPane2.getViewport().add(jTextArea2);
   jScrollPane1.getViewport().add(jTextArea1);
```

```
    }

    public void jButton1_actionPerformed(ActionEvent e) {
        jTextArea1.setText("======== Debut [Compilation Test]
                                                    ========\n");
        try {
            Scanner scanner = new Scanner(new
FileInputStream(jTextField1.getText()));
            scanner.next();
            while (scanner.symbol != ScannerTest.EOF) {
                jTextArea2.setText(scanner.err);
                jTextArea1.append("[Ligne:" + scanner.ligne + "] ");
                    jTextArea1.append(ScannerTest.representation(scanner.symbol));
                if ((scanner.symbol == ScannerTest.NUM) ||
                        (scanner.symbol == ScannerTest.ID) ||
                        (scanner.symbol == ScannerTest.SUP) ||
                        (scanner.symbol == ScannerTest.INF) ||
                        (scanner.symbol == ScannerTest.SUP_EGL) ||
                        (scanner.symbol == ScannerTest.INF_EGL) ||
                        (scanner.symbol == ScannerTest.EGL) ||
                        (scanner.symbol == ScannerTest.DIF) ||
                        (scanner.symbol == ScannerTest.IF) ||
                        (scanner.symbol == ScannerTest.THEN) ||
                        (scanner.symbol == ScannerTest.ELSE) ||
                        (scanner.symbol == ScannerTest.FOR) ||
                        (scanner.symbol == ScannerTest.WHILE) ||
                        (scanner.symbol == ScannerTest.EXTERN) ||
                        (scanner.symbol == ScannerTest.INT) ||
                        (scanner.symbol == ScannerTest.VOID) ||
                        (scanner.symbol == ScannerTest.RETURN)) {
                    jTextArea1.append("(" + scanner.chars + ")\n");}
                 else {
                    jTextArea1.append("\n");}
                scanner.next();
            }
            scanner.close();
        } catch (IOException ex) {
            //jTextArea1.append(ex);
            System.exit(-1);
        }

        jTextArea1.append("======== Fin [compilation Test] ========\n");

    }
```

```
    public void jTextField1_actionPerformed(ActionEvent e) {

    }

}


class compilateur_jTextField1_actionAdapter implements ActionListener {
    private compilateur adaptee;
    compilateur_jTextField1_actionAdapter(compilateur adaptee) {
        this.adaptee = adaptee;
    }

    public void actionPerformed(ActionEvent e) {
        adaptee.jTextField1_actionPerformed(e);
    }
}



class compilateur_jButton1_actionAdapter implements ActionListener {
    private compilateur adaptee;
    compilateur_jButton1_actionAdapter(compilateur adaptee) {
        this.adaptee = adaptee;
    }

    public void actionPerformed(ActionEvent e) {
        adaptee.jButton1_actionPerformed(e);
    }
}
```

Lexical.java

```
package compilateur;

import java.awt.Toolkit;
import javax.swing.SwingUtilities;
import javax.swing.UIManager;
import java.awt.Dimension;
```

```
public class Lexical {
   boolean packFrame = false;

   public Lexical() {
      compilateur frame = new compilateur();
  if (packFrame) {
         frame.pack();
      } else {
         frame.validate();
      }

    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
      Dimension frameSize = frame.getSize();
      if (frameSize.height > screenSize.height) {
         frameSize.height = screenSize.height;
      }
      if (frameSize.width > screenSize.width) {
         frameSize.width = screenSize.width;
      }
      frame.setLocation((screenSize.width - frameSize.width) / 2,
                  (screenSize.height - frameSize.height) / 2);
      frame.setVisible(true);
   }


   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            try {
               UIManager.setLookAndFeel(UIManager.
                              getSystemLookAndFeelClassName());
            } catch (Exception exception) {
               exception.printStackTrace();
            }

            new Lexical();
         }
      });
   }
}
```

Scanner.java

```
package compilateur;

import java.io.*;

public class Scanner implements Token {

   // dernier symbole (lexeme) qui a ete lu (voir Token)
   public int symbol;

   // position

        public int ligne = 1;
      public String err ="";
   // representation en characteres du dernier symbole
   public String chars;

   // le flux de characteres
   private InputStream in;

   // prochain charactere
   private char c;

   // tampon pour assembler des characteres
   private StringBuffer buf = new StringBuffer();

   // constructeur
   // premier charactere deja lu au bout de la construction
   Scanner(InputStream in) {
        this.in = in;
        nextChar();
   }

   // traitement d'erreur
   public String error(String msg) {
        return err += (msg+"\n");
           }


//////////////////////////////////////////////////////////////////////
```

```
// lit le prochain charactere
private void nextChar() {
    try {
        c = (char)in.read();
    } catch(IOException ex) {
        error("Erreur de lecture : " + ex.toString());
    }
}

////////////////////////////////////////////////////////////////////

// lit le prochain lexeme
// celui-ci est stocke dans symbol, son contenu dans chars
public void next() {

    // espaces blancs
    while (c <= ' ') {
    //ligne
    if(c =='\n') this.ligne += 1;
        nextChar();
    }

    switch(c) {

            // nombres
    case '0': case '1': case '2': case '3': case '4': case '5': case '6':
    case '7': case '8': case '9':
            buf.setLength(0);
        buf.append(c); nextChar();
        while ('0' <= c && c <= '9'){
            buf.append(c); nextChar();
        }
        chars = buf.toString();
            symbol = NUM;
            break;
            // mots cle et identificateurs
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
    case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':
    case 'o': case 'p': case 'q': case 'r': case 's': case 't': case 'u':
    case 'v': case 'w': case 'x': case 'y': case 'z':
    case 'A': case 'B': case 'C': case 'D': case 'E': case 'F': case 'G':
    case 'H': case 'I': case 'J': case 'K': case 'L': case 'M': case 'N':
    case 'O': case 'P': case 'Q': case 'R': case 'S': case 'T': case 'U':
```

```
case 'V': case 'W': case 'X': case 'Y': case 'Z':
   buf.setLength(0);
   buf.append(c); nextChar();
   while (('0' <= c && c <= '9') ||
          ('a' <= c && c <= 'z') ||
          ('A' <= c && c <= 'Z')) {
       buf.append(c); nextChar();
   }

   chars = buf.toString();

   // mots cle if, then, else, ...
   if (chars.equals("if")) symbol = IF;
   else if (chars.equals("then")) symbol = THEN;
        else if (chars.equals("else")) symbol = ELSE;
        else if (chars.equals("for")) symbol = FOR;
        else if (chars.equals("while")) symbol = WHILE;
        else if (chars.equals("extern")) symbol = EXTERN;
        else if (chars.equals("int")) symbol = INT;
        else if (chars.equals("void")) symbol = VOID;
        else if (chars.equals("return")) symbol = RETURN;

   else
   // identificateur
   symbol = ID;

   break;

   // operateurs numeriques
case '+':
   symbol = PLUS; nextChar(); break;
case '-':
   symbol = MOINS; nextChar(); break;
case '*':
   symbol = MULT; nextChar(); break;
case '/':
  buf.setLength(0);
  buf.append(c); nextChar();
     if(c == '*') {
        nextChar();
       while ((c != '*') && (c != (char) -1) )
        {   buf.append(c);
            nextChar();}
        if(c != '*'){
```

```
            nextChar();
            error("commentaire non terminé: " + " (ligne :"+ligne+")");
            symbol = ERROR;}
          else
          nextChar();
          if(c == '/') {symbol = COMENT;
                    }
                         }
       else

     symbol = DIV; nextChar();
break;

  case '<':
          buf.setLength(0);
     buf.append(c); nextChar();
          if(c == '=') {symbol = INF_EGL;
                                  buf.append(c);
                                  nextChar();}
          else
          if(c == '<') {symbol = DEC_GH;
                                  buf.append(c);
                                  nextChar();}
          else symbol = INF;

          chars = buf.toString();
          break;
  case '>':
          buf.setLength(0);
     buf.append(c); nextChar();
          if(c == '=') {symbol = SUP_EGL;
                                  buf.append(c);
                                  nextChar();}
          else
          if(c == '>') {symbol = DEC_DR;
                                  buf.append(c);
                                  nextChar();}
          else symbol = SUP;
          chars = buf.toString();
          break;
  case '!':
          buf.setLength(0);
     buf.append(c); nextChar();
          if(c == '=') {symbol = DIF;
```

```
                                    buf.append(c);
                                    nextChar();}
        else
        symbol = NON_LOG;
        chars = buf.toString();
        break;
case '&':
        buf.setLength(0);
   buf.append(c); nextChar();
        if(c == '&') {symbol = ET_LOG;
                                    buf.append(c);
                                    nextChar();}
        else
        symbol = ET_BIN;
        chars = buf.toString();
        break;
case '|':
        buf.setLength(0);
   buf.append(c); nextChar();
        if(c == '|') {symbol = OU_LOG;
                                    buf.append(c);
                                    nextChar();}
        else
        symbol = OU_BIN;
        chars = buf.toString();
        break;
case '=':
        buf.setLength(0);
   buf.append(c); nextChar();
        if(c == '=') {symbol = EGL;
                                    buf.append(c);
                                    chars = buf.toString();
                                    nextChar();}
        else
   error("charactere incorrect: " +"="+c + " (ligne :"+ligne+")");
   break;

   // delimiteurs (, ), {, }, ;,
case '(':
   symbol = PAR_OUV; nextChar(); break;
case ')':
   symbol = PAR_FER; nextChar(); break;
case '{':
   symbol = ACC_OUV; nextChar(); break;
```

```
        case '}':
           symbol = ACC_FER; nextChar(); break;
        case ';':
           symbol = PT_VRG; nextChar(); break;
        case ',':
           symbol = VRG; nextChar(); break;


        case (char) -1:
           symbol = EOF;
           break;
        default:
          error("charactere incorrect: " + c + " (ligne :"+ligne+")");
        }
    }

    // fermer flux d'entree
    public void close() throws IOException {
        in.close();
    }

}
```

ScannerTest.java

```
package compilateur;

import java.io.*;

public class ScannerTest implements Token {


    // representation
        public static String representation(int symbol) {
           switch(symbol) {
           case ERROR: return "<error>";
           case EOF: return "EOF";
```

```
        // valeurs
case NUM: return "NUM";
case ID: return "ID";

        // operateurs numeriques
case PLUS: return "PLUS";
case MOINS: return "MOINS";
case MULT: return "MULTIPLICATION";
case DIV: return "DIV";
    case ET_BIN: return "ET BINAIRE";
    case OU_BIN: return "OU BINAIRE";
    case DEC_DR: return "DECALAGE A DROITE";
    case DEC_GH: return "DECALAGE A GAUCHE";


    // operateurs booleen
    case NON_LOG: return "NON LOGIQUE";
    case ET_LOG: return "ET LOGIQUE";
    case OU_LOG: return "OU LOGIQUE";

    // operateurs de comparaison
    case INF: return "OPERATEUR INFERIEUR";
    case SUP: return "OPERATEUR SUPERIEUR";
    case INF_EGL: return "OPERATEUR INFERIEUR OU EGAL";
    case SUP_EGL: return "OPERATEUR SUPERIEUR OU EGAL";
    case EGL: return "OPERATEUR EGAL";
    case DIF: return "OPERATEUR DIFERENT";


    // mots cle
    case EXTERN: return "MOT CLE ";
    case INT: return "MOT CLE ";
    case VOID: return "MOT CLE ";
    case RETURN: return "MOT CLE ";
    case IF: return "MOT CLE ";
    case THEN: return "MOT CLE ";
    case ELSE: return "MOT CLE ";
    case FOR: return "MOT CLE ";
    case WHILE: return "MOT CLE ";

    // parentheses
    case PAR_OUV: return "PARENTHESE OUVRANTE";
    case PAR_FER: return "PARENTHESE FERMANTE";
    case ACC_OUV: return "ACCOLADE OUVRANTE";
```

```
            case ACC_FER: return "ACCOLADE FERMANTE";
            case COMENT: return "COMMENTAIRE";
            case VRG: return "VIRGULE";
            case PT_VRG: return "POINT VIRGULE";

        default: return "<INCONNU>";
        }
    }

}
```

Token.java

```
package compilateur;

interface Token {
   static final int

        / / symboles speciaux
        ERROR = 0,
        EOF = ERROR + 1,

        / / valeurs
        NUM = EOF + 1,
        ID = NUM + 1,

        / / operateurs numeriques
        PLUS = ID + 1,
        MOINS = PLUS + 1,
        MULT = MOINS + 1,
        DIV = MULT + 1,
        ET_BIN = DIV + 1,
        OU_BIN = ET_BIN + 1,
        DEC_DR = OU_BIN + 1,
        DEC_GH = DEC_DR + 1,

        / / operateurs booleen
```

```
        NON_LOG = DEC_GH + 1,
        ET_LOG = NON_LOG + 1,
        OU_LOG = ET_LOG + 1,


        / / operateurs de comparaison
        INF = OU_LOG + 1,
        SUP = INF + 1,
        INF_EGL = SUP + 1,
        SUP_EGL = INF_EGL + 1,
        EGL = SUP_EGL + 1,
        DIF = EGL + 1,

        / / mots cle
        EXTERN = DIF + 1,
        INT = EXTERN + 1,
        VOID = INT + 1,
        RETURN = VOID + 1,
        IF = RETURN + 1,
        THEN = IF + 1,
        ELSE = THEN + 1,
        FOR = ELSE + 1,
        WHILE = FOR + 1,


        / / parentheses
        PAR_OUV = WHILE + 1,
        PAR_FER = PAR_OUV + 1,
        ACC_OUV = PAR_FER + 1,
        ACC_FER = ACC_OUV + 1,
        COMENT = ACC_FER +1,
        VRG = COMENT + 1,
        PT_VRG = VRG + 1;
}
```

IV. Jeux d'essai :

Pour exécuter le programme il suffit de double cliquer sur l'archivage compilateur2.jar (ceci suppose le JDK installé sur la machine).

Après, il faut spécifier l'emplacement du fichier à scanner dans la zone chemin, puis cliquer sur compiler et c'est fait !