UNIVERSIDAD DE SAN CARLOS DE GUATEMALA.

CENTRO UNIVERSITARIO DE OCCIDENTE.

DIVISIÓN DE CIENCIAS DE LA INGENIERÍA.

INGENIERÍA EN CIENCIAS Y SISTEMAS.

LABORATORIO DE LENGUAJES Y COMPILADORES 1.

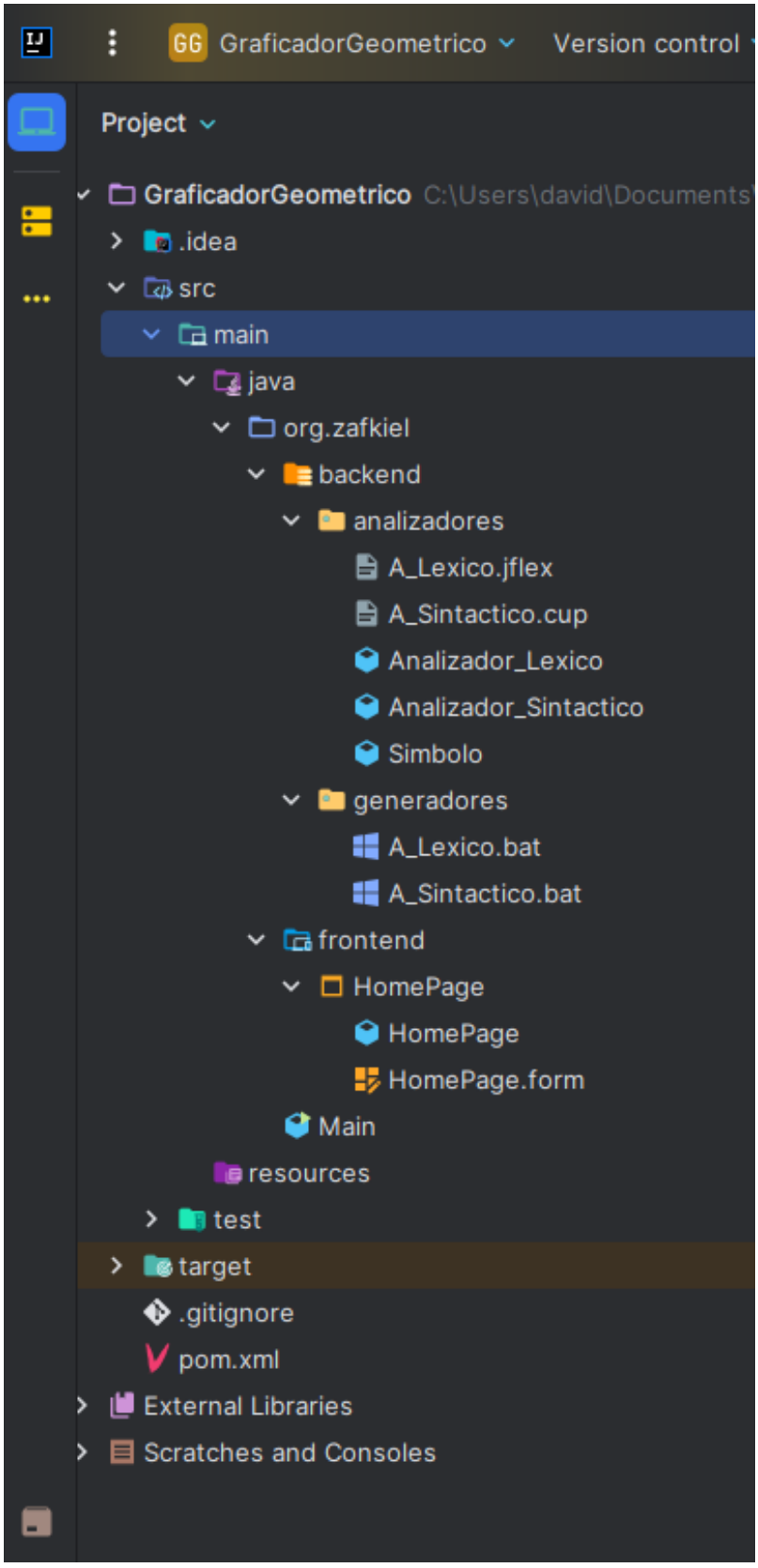
PRACTICA 1.

GRAFICADORGEOMETRICO

ESTUARDO DAVID BARRENO NIMATUJ.

CARNÉ: 201830233.

ESTRUCTURA DEL PROYECTO.



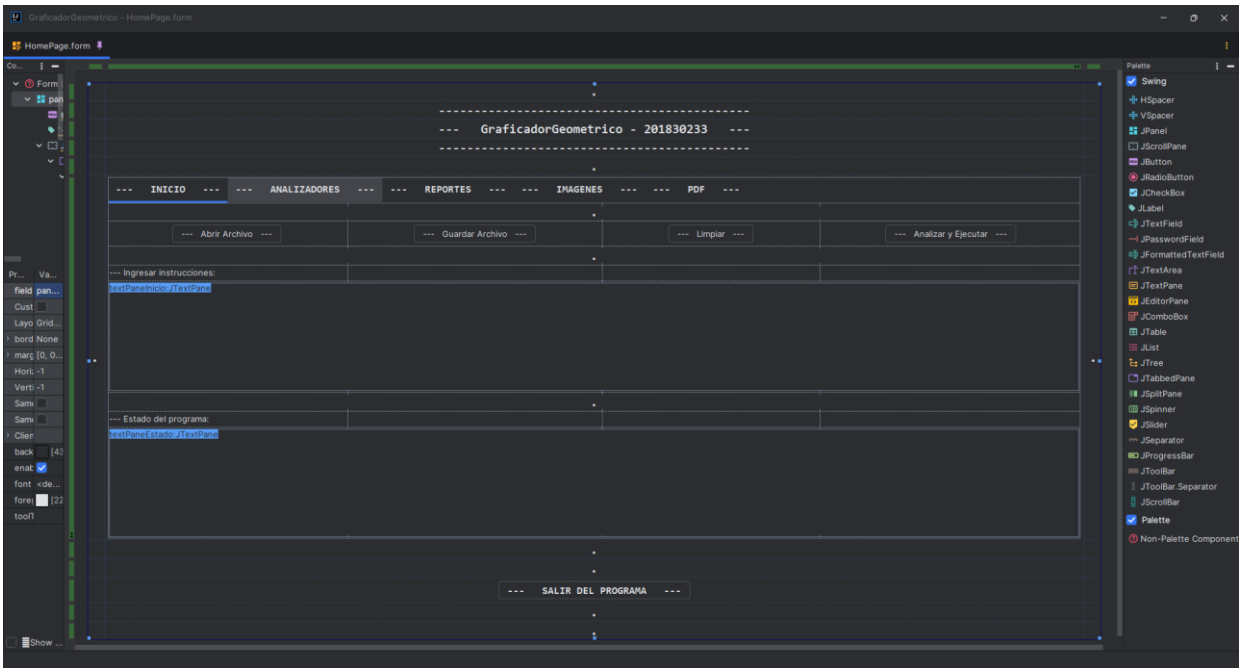
El proyecto cuenta con las diferentes clases y métodos dentro de la carpeta, asi mismo todos los archivos .bat, .jflex y .cup usados en el mismo se encuentran en sus respectivas categorías.

METODO MAIN.

```
public class Main {
    public static void main(String[] args) {
        HomePage homePage = new HomePage();
        homePage.setContentPane(homePage.panelHomePage);
        homePage.setExtendedState(JFrame.MAXIMIZED_BOTH);
        homePage.setUndecorated(true);
        homePage.setVisible(true);
    }
}
```

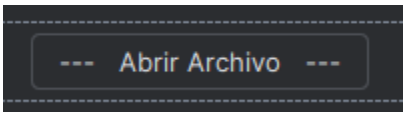
Este método simplemente llama al panel HomePage que es la interfaz gráfica base del proyecto.

METODO INTERFAZ GRAFICA.



La interfaz esta echa en un solo panel llamado HomePage, y tiene sub paneles y todos sus detalles diseñados en una sola clase.

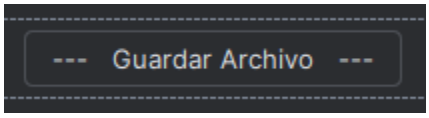
METODO PARA ABRIR ARCHIVOS TXT.



Este código permite al usuario seleccionar un archivo de texto, filtra para asegurar que solo se seleccionen archivos .txt, y si se selecciona uno válido, lo lee y muestra su contenido en un JTextPane. Si se selecciona un archivo que no es un .txt, se muestra un mensaje de error.

1. Se crea una instancia de JFileChooser, que es una clase de Swing que proporciona una interfaz gráfica para seleccionar archivos o directorios.
2. se crea un filtro (FileNameExtensionFilter) que permite mostrar solo archivos con la extensión .txt. Este filtro se establece en el JFileChooser para restringir la vista a los archivos de texto.
3. Se muestra el diálogo para abrir archivos. El método showOpenDialog retorna un valor entero que indica si el usuario seleccionó un archivo (JFileChooser.APPROVE_OPTION) o canceló la operación.
4. Se verifica que el archivo seleccionado tenga la extensión .txt. Esto asegura que solo se procesen archivos de texto, incluso si el usuario intenta seleccionar un archivo con una extensión diferente.
5. Si el archivo es un .txt, se procede a leer su contenido línea por línea usando un BufferedReader y un FileReader. Cada línea se añade a un StringBuilder, y al final se coloca todo el contenido en el JTextPane (textPaneInicio). Si ocurre un error durante la lectura del archivo, se captura y se imprime en la consola.

METODO PARA GUARDAR ARCHIVOS TXT.

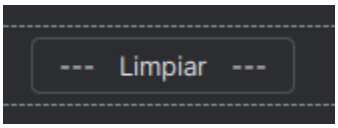


Este código permite al usuario guardar el contenido de un JTextPane en un archivo de texto. Se abre un cuadro de diálogo para seleccionar

la ubicación y el nombre del archivo, se asegura que el archivo tenga la extensión .txt, y luego guarda el contenido.

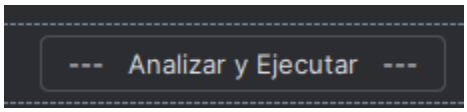
1. Se crea una instancia de JFileChooser, que es una ventana que permite al usuario elegir dónde y cómo guardar un archivo. Luego, se configura un filtro para que solo se muestren archivos con la extensión .txt.
2. Si el usuario eligió un archivo y presionó "Guardar", el archivo seleccionado se almacena en fileToSave.
3. Si el nombre del archivo seleccionado no termina con .txt, se agrega esta extensión automáticamente para asegurar que el archivo sea guardado como un archivo de texto.
4. Se utiliza un BufferedWriter junto con un FileWriter para escribir el contenido del JTextPane (textPaneInicio) en el archivo seleccionado. La función textPaneInicio.write(writer) se encarga de escribir el contenido completo en el archivo.

METODO LIMPIAR.



Este método se encarga de limpiar el área de trabajo para introducir nuevas instrucciones.

METODO ANALIZAR Y EJECUTAR.



Este método se encarga de lo mas importante, extra el texto en el textPaneInicio, y lo manda a las reglas léxicas y semánticas de los analizadores Analizador_Lexico.jflex y Analizador_Sintactico.cup, en donde se realizan todos los análisis correspondientes. Se llaman los resultados del análisis y se procede a llenar los reportes.

A_LEXICO.JFLEX

Este código es parte de un proceso de análisis léxico para un compilador o intérprete, encargándose de identificar y clasificar las distintas partes de una entrada de código según su función y sintaxis dentro del lenguaje definido.

```
/*---   1era Area: Codigo Usuario   ---*/
//---> Paquetes, importaciones
package org.zafkiel.backend.analizadores;
import java_cup.runtime.*;
import java.util.List;
import java.util.ArrayList;
```

En esta sección se define donde se encuentra el archivo Analizador_Lexico.jflex y sus respectivas importaciones.

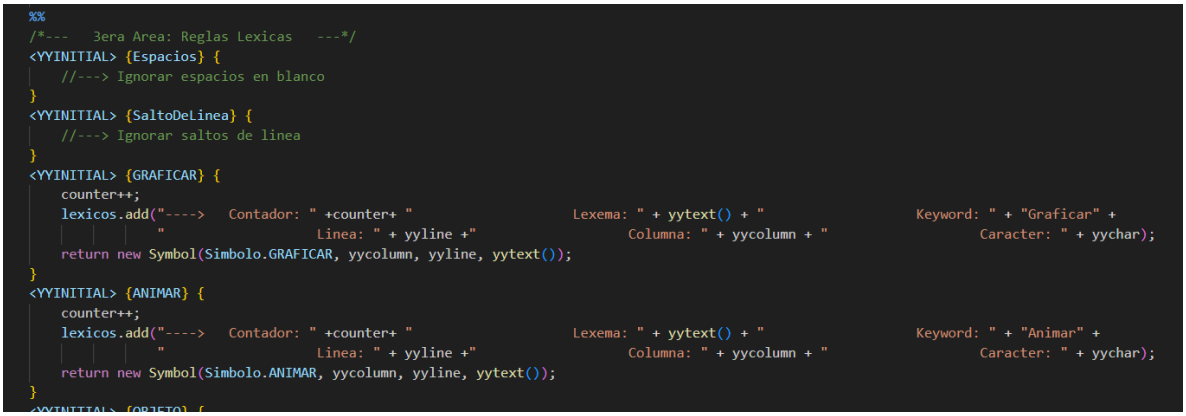
```
/*---   2da Area: Opciones y Declaraciones   ---*/
%%
//---> Directivas
%public
%class Analizador_Lexico
%cupsym Simbolo
%cup
//---> Estados
%{
    private int counter;
    private int counter2;
    private static List<String> lexicos = new ArrayList<>();
    public static List<String> getLexicos() {
        return lexicos;
    }
    private static List<String> errores = new ArrayList<>();
    public static List<String> getErrores() {
        return errores;
    }
%}
```

En esta sección se definen las directivas que usara el analizador, asi mismo el nombre de la clase que generara, y el tipo de symbolos a pasar al analizador de cup.

En los estados definimos variables y las listas que usaremos para acceder a la clasificaion de los tokens analizados.

```
%full
//---> Expresiones Regulares
    Espacios = " "
    SaltoDeLinea = \n|\r|\r\n
    DIGITO = [0-9]
    LETRA = [A-Za-z]
    IDENTIFICADOR = [a-zA-Z_][a-zA-Z_0-9]*
    ENTERO = {DIGITO}+
    DECIMAL = {DIGITO}+(\.{DIGITO}+)?
    GRAFICAR = "graficar"
```

En esta sección de Expresines Regulares, definiremos todas las palabras reservadas y las expresiones regulares que nos serán de utilidad en el análisis.



```
%%
/*---   3era Area: Reglas Lexicas   ---*/
<YYINITIAL> {Espacios} {
    //---> Ignorar espacios en blanco
}
<YYINITIAL> {SaltoDeLinea} {
    //---> Ignorar saltos de linea
}
<YYINITIAL> {GRAFICAR} {
    counter++;
    lexicos.add("---->   Contador: " +counter+ "                  Lexema: " + yytext() + "                  Keyword: " + "Graficar" +
                "                  Linea: " + yyline +"                  Columna: " + yycolumn + "                  Caracter: " + yychar);
    return new Symbol(Simbolo.GRAFICAR, yycolumn, yyline, yytext());
}
<YYINITIAL> {ANIMAR} {
    counter++;
    lexicos.add("---->   Contador: " +counter+ "                  Lexema: " + yytext() + "                  Keyword: " + "Animar" +
                "                  Linea: " + yyline +"                  Columna: " + yycolumn + "                  Caracter: " + yychar);
    return new Symbol(Simbolo.ANIMAR, yycolumn, yyline, yytext());
}
```

En esta sección de Reglas Lexicas, definiremos que hacer cuando se encuentre alguna palabra reservada o expresión regular, en este caso las clasificamos principalmente en 2 grupos.

- Para el análisis lexico, usando una ArrayList agregamos todos los tokens encontados junto con sus características.
- Para el análisis Sintactico, usando el `retutn new Symbol` que son los valores que pasaremos a cup, solo le definimos el nombre del token con el que lo vamos a reconocer en cup `Simbolo.GRAFICAR`, en este caso lo uso en mayúsculas para no confundirme mas adelante con las reglas de cup.

```
<YYINITIAL> . {
    counter++;
    counter2++;
    lexicos.add("---->   Contador: " +counter+ "                Lexema: " + yytext() + "                Keyword: " + "ERROR" +
                "                Linea: " + yyline +"                Columna: " + yycolumn + "                Caracter: " + yychar);
    errores.add("---->   Contador: " +counter2+ "                Lexema: " + yytext() + "                Keyword: " + "ERROR - Simbolo no existe en el lenguaje" +
                "                Linea: " + yyline +"                Columna: " + yycolumn + "                Caracter: " + yychar);

}
```

En el caso de los ERRORES, los agregamos tanto en el ArrayList de tokens clasificados como en otro ArrayList en donde van solo los Errores léxicos y sintácticos, para poder representarlos en los reportes de mejor manera.

A_SINTACTICO.CUP

```
/*---   1era Area: Codigo Usuario   ---*/
//---> Paquetes, importaciones
package org.zafkiel.backend.analizadores;
import java_cup.runtime.Symbol;
import java.util.List;
import java.util.ArrayList;
import java.io.FileWriter;
import java.io.IOException;
```

En esta sección definimos los paquetes e importaciones que usaremos mas adelante.

```
9
10  //---> Codigo para el parser, variables, metodos
11  parser code
12  {:
13      public String resultado = "";
14      public String resultado2 = "";
15      public String resultado3 = "";
16      public String resultado4 = "";
17      public String resultado5 = "";
18      public String color = "";
19      public int colorAzul = 0;
20      public int colorRojo = 0;
```

En esta sección del código para el parser definimos variables y métodos que usaremos, en este caso dejo muchas variables en public, ya que accederé a ellas después para los reportes.

```
private static List<String> instruccionSintactica = new ArrayList<>();
public static List<String> getInstruccionSintactica() {
    return instruccionSintactica;
}
private static List<String> sintacticos = new ArrayList<>();
public static List<String> getSintacticos() {
    return sintacticos;
}
```

También use la estructura de ArrayList para almacenar las reglas o producciones validas como errores de las cadenas a analizar, asi es mas fácil para los reportes.

```
public void syntax_error(Symbol s){
    String lexema = s.value.toString();
    int fila = s.right;
    int columna = s.left;
    sintacticos.add("--->   ERROR SINTACTICO RECUPERADO!!!
}

public void unrecovered_syntax_error(Symbol s){
String lexema = s.value.toString();
int fila = s.right;
int columna = s.left;
sintacticos.add("--->   ERROR SINTACTICO MODO PANICO!!!
```

Se usan 2 tipos de Errores en el análisis sintáctico, El ERROR SINTACTICO RECUPERADO, que quiere decir que hubo un error y no afecto en el análisis, por lo tanto, se recuperó y pudimos continuar.

El ERROR SINTACTICO MODO PANICO, aca según entiendo son errores que no se pueden recuperar y por lo tanto detienen la ejecución del programa, pero al no saber como recuperarme de estos errores no realizan una acción especifica solo los almaceno y continuo con el análisis que no tenga este error.

```
/*---    2da Area: Declaraciones    ---*/
//---> Terminales
terminal GRAFICAR, ANIMAR, OBJETO, ANTERIOR, LINEA, CURVA, CI
terminal String ENTERO, DECIMAL, IDENTIFICADOR;
```

Aquí se definen los terminales, que son los tokens que el analizador léxico reconocerá. Incluyen palabras clave, operadores, y tipos de datos.

La razón de definir como “String” ENTERO, DECIMAL, IDENTIFICADOR, es porque al momento de guardar los valores en las variables y listas, evitamos errores, en dado caso que se necesiten hacer operaciones con estos valores, los podemos pasar a Double y operar, depues se regresan a String y seguimos con la ejecución normal.

```
//---> No Terminales
non terminal String INICIO, CADENA, EXPRESIONES, INSTRUCCION, FIGURA, EXPRESION,

//---> Precedencia de Menor a Mayor
precedence left MAS, MENOS;
precedence left POR, DIV;
precedence left PARA, PARB;
```

En esta sección definimos en “non terminal” los no terminales representan los distintos componentes de la gramática, como la estructura general del programa y las expresiones.

Define la precedencia de los operadores matemáticos.

```
/*---   3era Area: Reglas Semanticas   ---*/
start with INICIO;


INICIO ::= CADENA
        ;
//   |
CADENA ::= CADENA EXPRESIONES
    | EXPRESION
    | CADENA EXPRESIONES3
    | CADENA error{: sintacticos.add("--->   Error en la instrucción, continuando con la siguiente.\n"); :}
    | error {: sintacticos.add("--->   Error en la instrucción, continuando con la siguiente.\n"); :}
;
```

Las reglas definen cómo se combinan los terminales y no terminales para formar expresiones válidas en el lenguaje.

INICIO: La regla principal que comienza con `INICIO`, que puede ser una `CADENA`.

CADENA: Define cómo las cadenas de instrucciones pueden ser formadas por expresiones o errores.

EXPRESIONES3: Maneja la animación de objetos.

EXPRESIONES: Maneja la instrucción para graficar círculos, cuadrados, rectángulos, líneas, y polígonos.

GRAFICAR CIRCULOS.

Círculo

Para graficar un círculo se usa la instrucción:

graficar circulo (<nombre>, <posx>, <posy>, <radio>, <color>)

ejemplo:

graficar circulo (figura_1, 25 + 12, 25, 15/3, rojo)

Definimos nuestra producción:

```
EXPRESIONES ::= GRAFICAR CIRCULO PARA IDENTIFICADOR:i COMA OPERADORES COMA
OPERADORES2 COMA OPERADORES3 COMA COLOR PARB
```

Los únicos no terminales son las operaciones que se pueden realizar en los OPERADORES, y COLOR, porque espera uno de todos los colores validos.

```
String contenidoDOT = "graph G { layout=neato;" + i + " [shape=circle,
pos=\"" + resultado + "," + resultado2 + "!\", width=" + radio3 + ", color="
+ color + ", style=filled, label=" + i + "];point1 [shape=point,
pos=\"0,0!\", width=0.1, color=black];}";
```

Dentro de esta producción si se valida, podemos graficar usando la herramienta Graphiz.

ContenidoDOT: lleva la regla o instrucción para graficar un circulo con los parámetros solicitados, los parámetros los dejamos en forma de variables, ya que se espera usar la misma instrucción varias veces.

```
try {
            // Escribimos el contenido en un archivo DOT
            FileWriter writer = new FileWriter(i+".dot");
            writer.write(contenidoDOT);
            writer.close();
            estadoGraphiz.add("\n--->   Archivo DOT "+i+".dot generado correctamente.");
            try {
            // Ejecutamos el comando de Graphviz para generar la imagen desde el archivo DOT
            ProcessBuilder processBuilder = new ProcessBuilder("dot", "-Tpng", i+".dot", "-o", i+".png");
            Process process = processBuilder.start();
            int exitCode = process.waitFor();
            if (exitCode == 0) {
                estadoGraphiz.add("\n--->   Imagen "+i+".png generada correctamente.");
            } else {
                estadoGraphiz.add("\n--->   ERROR al generar la imagen Código de salida: " + exitCode);
            }
        } catch (IOException | InterruptedException e) {
            estadoGraphiz.add("\n--->   ERROR al generar la imagen: " + e.getMessage());
        }
```

Creamos el archivo .dot: usando el procesbuilder de java podemos mandar el comando a consola de la creación de un archivo .dot.

Creamos la imagen PNG que contiene la isntruccion dada de la grafica del circulo.

```
try {
    // Ejecutamos el comando de Graphviz para generar el pdf desde el archivo DOT
    ProcessBuilder processBuilder = new ProcessBuilder("dot", "-Tpdf", i+".dot", "-o", i+".pdf");
    Process process = processBuilder.start();
    int exitCode = process.waitFor();
    if (exitCode == 0) {
        estadoGraphiz.add("\n--->   PDF "+i+".pdf generada correctamente.");
    } else {
        estadoGraphiz.add("\n--->   ERROR al generar el pdf Código de salida: " + exitCode);
    }
} catch (IOException | InterruptedException ex) {
    estadoGraphiz.add("\n--->   ERROR al generar el pdf: " + ex.getMessage());
}
} catch (IOException e) {
    estadoGraphiz.add("\n--->   ERROR al escribir el archivo DOT: " + e.getMessage());
}
objetoCirculo++;
```

Usando la misma lógica podemos crear conjuntamente el archivo PDF que contiene la instrucción del circulo dada. Podemos repetir el proceso para todas las figuras a graficar.

GRAFICAR CUADRADOS.

Cuadrados

Para graficar un cuadrado se usa la instrucción:

graficar cuadrado (<nombre>, <posx>, <posy>, <tamaño lado>, <color>)

ejemplo:

graficar cuadrado (figura_cuadrada, 12*3, 15+1, (15-3) / 4, verde)

Definimos nuestra producción:

```
GRAFICAR CUADRADO PARA IDENTIFICADOR:i COMA OPERADORES COMA OPERADORES2 COMA
OPERADORES3 COMA COLOR PARB
```

GRAFICAR RECTANGULOS.

Rectángulo

Para graficar un rectángulo se usa la instrucción:

graficar rectangulo (<nombre>, <posx>, <posy>, <ancho>, <alto>, <color>)

ejemplo:

graficar rectangulo (rectangulo_12, 12 * 3 + 2, 15, 4 / 4, negro)

Definimos nuestra producción:

```
GRAFICAR RECTANGULO PARA IDENTIFICADOR:i COMA OPERADORES COMA OPERADORES2
COMA OPERADORES3 COMA OPERADORES4 COMA COLOR PARB
```

GRAFICAR LINEAS.

Línea

Para graficar un rectángulo se usa la instrucción:

graficar linea (<nombre>, <posx1>, <posy1>, <posx2>, <posy2>, <color>)

ejemplo:

graficar linea (linea_Amarillo, 12 * 3 + 2, 15, 4 / 4, 50 * 1, amarillo)

Definimos nuestra producción:

```
GRAFICAR LINEA PARA IDENTIFICADOR:i COMA OPERADORES COMA OPERADORES2 COMA
OPERADORES3 COMA OPERADORES4 COMA COLOR PARB
```

GRAFICAR POLIGONOS.

Polígono

Para graficar un poligono se usa la instrucción:

graficar poligono (<nombre>, <posx>, <posy>, <cantidad lados>, <ancho>, <alto>, <color>)

ejemplo:

graficar poligono (PoligA, 12 + 2, 15, 6, 50 / 2, 12, amarillo)

Definimos nuestra producción:

```
GRAFICAR POLIGONO PARA IDENTIFICADOR:i COMA OPERADORES COMA OPERADORES2 COMA
OPERADORES3 COMA OPERADORES4 COMA OPERADORES5 COMA COLOR PARB
```

GRAFICAR PUNTO DE ORIGEN.

```
point1 [shape=point, pos=\"0,0!\", width=0.1, color=black];
```

Este punto “point1” esta en el origen, lo grafico con todas las imágenes, ya que asi podemos ver si de verdad se están graficando en las coordenadas indicadas y siempre respetando la escala.

```
OPERADORES ::= EXPRESIONES2:a {: resultado = a; :}
            ;
OPERADORES2 ::= EXPRESIONES2:a {: resultado2 = a;     :}
            ;
OPERADORES3 ::= EXPRESIONES2:a {: resultado3 = a;     :}
            ;
OPERADORES4 ::= EXPRESIONES2:a {: resultado4 = a;     :}
            ;
OPERADORES5 ::= EXPRESIONES2:a {: resultado5 = a;     :}
            ;
EXPRESIONES2 ::= EXPRESIONES2:a E:b {: RESULT = a + "\n" + b; :}
               | E:a                          {: RESULT = a; :}
```

OPERADORES Y RESULTADOS.

Los operadores no son mas que las posibles sumas, restas, multiplicaciones y divisiones que pueden venir dentro de los parámetros de las figuras a graficar, son 5 porque 5 es el máximo de posibles parámetros difenretes que puede tener una figura (Poligono), la razón de asignarle las variables "resultado" es para poder diferenciar los parámetros finales de cada figura sin confundir las variables, al tenerlas en variables diferentes, las puedo llamar desde otra producción, como la de agregarlas a las ArrayList para los reportes, como para definir los parámetros en el archivo DOT.

PRODUCCIONES SUMA.

```
E::= E:a MAS E:b {:
                    double val1 = Double.parseDouble(a);
                    double val2 = Double.parseDouble(b);
                    double r = val1 + val2;
                    RESULT = String.valueOf(r);
                    reporteMatematicos.add("--->   Suma                Ocurrencia: "+val1+"+"+val2); operacionSuma++;
                    :}
```

PRODUCCIONE S RESTA.

```
|E:a MENOS E:b {:
                double val1 = Double.parseDouble(a);
                double val2 = Double.parseDouble(b);
                double r = val1 - val2;
                RESULT = String.valueOf(r);
                reporteMatematicos.add("--->   Resta               Ocurrencia: "+val1+"-"+val2);    operacionResta++;
                :}
```

PRODUCCIONES DIVISION.

```
|E:a DIV E:b {:
              double val1 = Double.parseDouble(a);
              double val2 = Double.parseDouble(b);
              double r = val1 / val2;
              RESULT = String.valueOf(r);
              reporteMatematicos.add("--->   Division               Ocurrencia: "+val1+"/"+val2); operacionDiv++;
              :}
```

PRODUCCIONES MULTIPLICACION.

```
|E:a POR E:b {:
                double val1 = Double.parseDouble(a);
                double val2 = Double.parseDouble(b);
                double r = val1 * val2;
                RESULT = String.valueOf(r);
                reporteMatematicos.add("--->   Multiplicacion                Ocurrencia: "+val1+"*"+val2);   operacionPor++;
                :}
```

INTERPRETACION DE DOUBLE E INT JUNTOS.

```
                reporteMatematicos.add("
                :}
    |PARA E:a {:     RESULT = a; :} PARB
    |ENTERO:a {:     RESULT = a; :}
    |DECIMAL:a {:    RESULT = a; :}
    ;
```

Lo que hice fue pasar todo a double, ya sea que venga en double o en entero, asi no hay problema entre las operaciones con decimales.

PRODUCCION COLOR

```
    ;
COLOR ::= BLUE {:   color = "blue"; colorAzul++;    :}| RED {:   color = "red"; colorRojo++;    :}
| YELLOW {:   color = "yellow"; colorAmarillo++;    :}| GREEN {:   color = "green"; colorVerde++;   :}
| PINK {:   color = "pink"; colorRosado++;  :}| ORANGE {:   color = "orange"; colorAnaranjado++;    :}
| PURPLE {:   color = "purple"; colorMorado++;  :}| DEEPSKYBLUE {:   color = "deepskyblue"; colorCeleste++; :}
| GOLD {:   color = "gold"; colorDorado++;  :}
;
```

Acá definimos los colores que pueden aceptar las producciones únicamente.

REPORTES.

La razón de usar varias ArrayList y diferentes Variables es para que al momento de presentar los reportes, ya tenga clasificado en donde va cada uno.

Reporte por ArrayList.

```
List<String> lexicos = Analizador_Lexico.getLexicos();
if (!lexicos.isEmpty()) {
    for (String lexico2 : lexicos) {
        textPaneReporteLexico.setText(textPaneReporteLexico.getText()+lexico2+"\n");
    }
    lexicos.clear();
}
```

Llamamos el ArrayList y como ya habíamos definido la estructura de la clasificación, solo debemos copiar esa misma en el lugar de cada reporte.

Reporte por variable:

```
textPaneReporteColores.setText(textPaneReporteColores.getText()+
        "\n\n--->   El color AZUL se a usado: "+String.valueOf(sintactico.colorAzul)+" veces.\n\n"+
        "--->   El color ROJO se a usado: "+String.valueOf(sintactico.colorRojo)+" veces.\n\n"+
        "--->   El color AMARILLO se a usado: "+String.valueOf(sintactico.colorAmarillo)+" veces.\n\n"+
        "--->   El color VERDE se a usado: "+String.valueOf(sintactico.colorVerde)+" veces.\n\n"+
        "--->   El color ROSADO se a usado: "+String.valueOf(sintactico.colorRosado)+" veces.\n\n"+
        "--->   El color ANARANJADO se a usado: "+String.valueOf(sintactico.colorAnaranjado)+" veces.\n\n"+
        "--->   El color MORADO se a usado: "+String.valueOf(sintactico.colorMorado)+" veces.\n\n"+
        "--->   El color CELESTE se a usado: "+String.valueOf(sintactico.colorCeleste)+" veces.\n\n"+
        "--->   El color DORADO se a usado: "+String.valueOf(sintactico.colorDorado)+" veces.\n\n");
```

Llamamos al valor final de cada variable y acomodamos el reporte a nuestro gusto.