



# PRACTICA NUMERO 1

22.2.2021

---

Carlos Benjamín Pac Flores

Centro Universitario de Occidente (CUNOC)

Ingeniería en Ciencias y Sistemas

Febrero 2021

<b>Introducción</b>	<b>2</b>
<b>Plataforma de Desarrollo:</b>	<b>3</b>
<b>Alcances Tecnicos:</b>	<b>3</b>
<b>Requisitos De Software:</b>	<b>3</b>
<b>Detalle del Sistema:</b>	<b>4</b>
Estructura del analisis Lexico:	4
<b>Estructura del Análisis Sintáctico:</b>	<b>5</b>
Símbolos Terminales:	5
Símbolos No Terminales:	5
Reglas de Producción:	5
<b>Diagrama de Clases</b>	<b>8</b>
<b>Organización del Proyecto</b>	<b>10</b>



## Introducción

El siguiente documento da presenta una descripción de la estructura del software de la aplicación desarrollada para el entorno de Android, que por medio de una serie de instrucciones el software es capaz de generar figuras tales como, cuadrados, rectángulos, círculos y polígonos, por el tipo de estructura del software es capaz de poder resolver las instrucciones por medio de un analizador lexico y sintactico y por medio de este último poder resolver parámetros basados en operaciones entre números tales como sumas, restas, multiplicaciones y divisiones y agrupaciones.

## Plataforma de Desarrollo:

- Lenguaje Java en su versión 11 LTS ORACLE.
- Lenguaje Java en su distribución OPEN-JDK
- Sistema de gestión de proyecto Grandle.
- Android Studio
- Librería para análisis léxico JFLEX
- Librería para análisis sintáctico CUP

## Alcances Tecnicos:

Graficar figuras por medio de las librerías que dispone el entorno de Android, resolviendo mediante expresiones regulares y solución de gramática los atributos de dichas imágenes, así también mostrando dichos errores léxicos y sintácticos al momento de realizar las instrucciones para el programa.

## Requisitos De Software:

- Android en su version minima 4.2
- Ram recomendada 3 GB

## Detalle del Sistema:

### Estructura del analisis Lexico:

Para el funcionamiento del analizador léxico se establecen las siguientes expresiones regulares para el reconocimiento de los TOKENS del texto.

- **LineTerminator** = `[\r|\n|\r\n]+`  
Reconoce los fines de línea en el texto, como retornos de carro y nuevas líneas.
- **WhiteSpace** = `[\t\n]+`  
Reconoce los espacios entre textos, como los ya mencionados espacio y las tabulaciones.
- **RecervateWords** = `(graficar | animar | objeto | anterior)`  
Reconoce ciertas palabras que son reservadas para el programa.
- **Colors** = `(azul | rojo | verde | amarillo | naranja | morado | cafe | negro)`  
Reconoce los colores a utilizar en las figuras
- **TypeAnimation** = `(linea | curva)`  
Reconoce el tipo de animación que realizara una figura
- **TypeShape** = `(circulo | cuadrado | rectangulo | linea | poligono)`  
Reconoce el tipo de figura que se desea graficar.
- **Numbers** = `[0-9]+`  
Reconoce las cadenas de dígitos y las establece como un número en concreto.
- **Words** = `[a-zA-Z]+`  
Reconoce un conjunto de cadenas de letras, dicho reconocimiento es utilizado al momento de cometer errores.
- **{ + }**  
Reconoce el símbolo de suma.
- **{ - }**  
Reconoce el símbolo de resta
- **{ \* }**

Reconoce el símbolo de multiplicación

- { / }

Reconoce el símbolo de división

- { ( }

Reconoce el paréntesis de apertura

- { ) }

Reconoce el paréntesis de cierre

- { , }

Reconoce el separador de texto denominado COMA

## Estructura del Análisis Sintáctico:

Símbolos Terminales:

**ERROR, RESERVATE\_WORD\_GRAPHIC, RESERVATE\_WORD\_ANIMATION, RESERVATE\_WORD\_OBJECT, RESERVATE\_WORD\_REFERENCE, COLOR, ANIMATION, SHAPE\_CIR, SHAPE\_CUA, SHAPE\_REC, SHAPE\_LIN, SHAPE\_POL, NUMBER, COMA, SUMA, RESTA, MULTI, DIV, PARENTECIS\_APERTURA, PARENTECIS\_CIERRE**

Símbolos No Terminales:

**s, t, y, var, shapeParams, expr, term, factor**

Reglas de Producción:

El símbolo no terminal "s" es donde se inicia la producción de gramática.

**s ::= RESERVATE\_WORD\_GRAPHIC shapeParams  
| error**

La producción anterior establece como debe ser el inicio de las instrucciones del programa estableciendo como continuación los parámetros de alguna figura en específico

**shapeParams ::=**

**SHAPE\_CIR PARENTECIS\_APERTURA var COMA var COMA var COMA COLOR  
PARENTECIS\_CIERRE y**

**| SHAPE\_CUA PARENTECIS\_APERTURA var COMA var COMA var COMA COLOR  
PARENTECIS\_CIERRE y**

```

| SHAPE_REC PARENTECIS_APERTURA var COMA var COMA var COMA var COMA
COLOR PARENTECIS_CIERRE y
| SHAPE_LIN PARENTECIS_APERTURA var COMA var COMA var COMA var COMA
COLOR PARENTECIS_CIERRE y
| SHAPE_POL PARENTECIS_APERTURA var COMA var COMA var COMA var COMA
var COMA COLOR PARENTECIS_CIERRE y
| error

```

La producción anterior establece las características de generación de cada figura en el programa, la cantidad de parámetros y tipo así como la espera de una instrucción complementaria que puede ser una animación de la figura a graficar.

```

y ::=
    RESERVATE_WORD_ANIMATION RESERVATE_WORD_OBJECT
    RESERVATE_WORD_REFERENCE PARENTECIS_APERTURA var COMA var COMA
    ANIMATION PARENTECIS_CIERRE t
| t

```

La producción anterior establece la instrucción para poder animar una figura en concreto

```

t ::= s
| λ

```

La producción anterior establece la continuidad o el fin de instrucciones

```

var ::= expr
| error

```

La producción anterior establece el un conjunto de expresiones numéricas.

```

expr ::= expr SUMA term
| expr RESTA term
| term

```

La producción anterior establece la relación que debe de haber entre la suma y resta de dígitos, obedeciendo a la precedencia de los signos aritméticos.

```

term ::= term MULTI factor
| term DIV factor
| factor

```

La producción anterior establece la relación para poder realizar una multiplicación o división.

```

factor ::= NUMBER
| PARENTECIS_APERTURA expr PARENTECIS_CIERRE

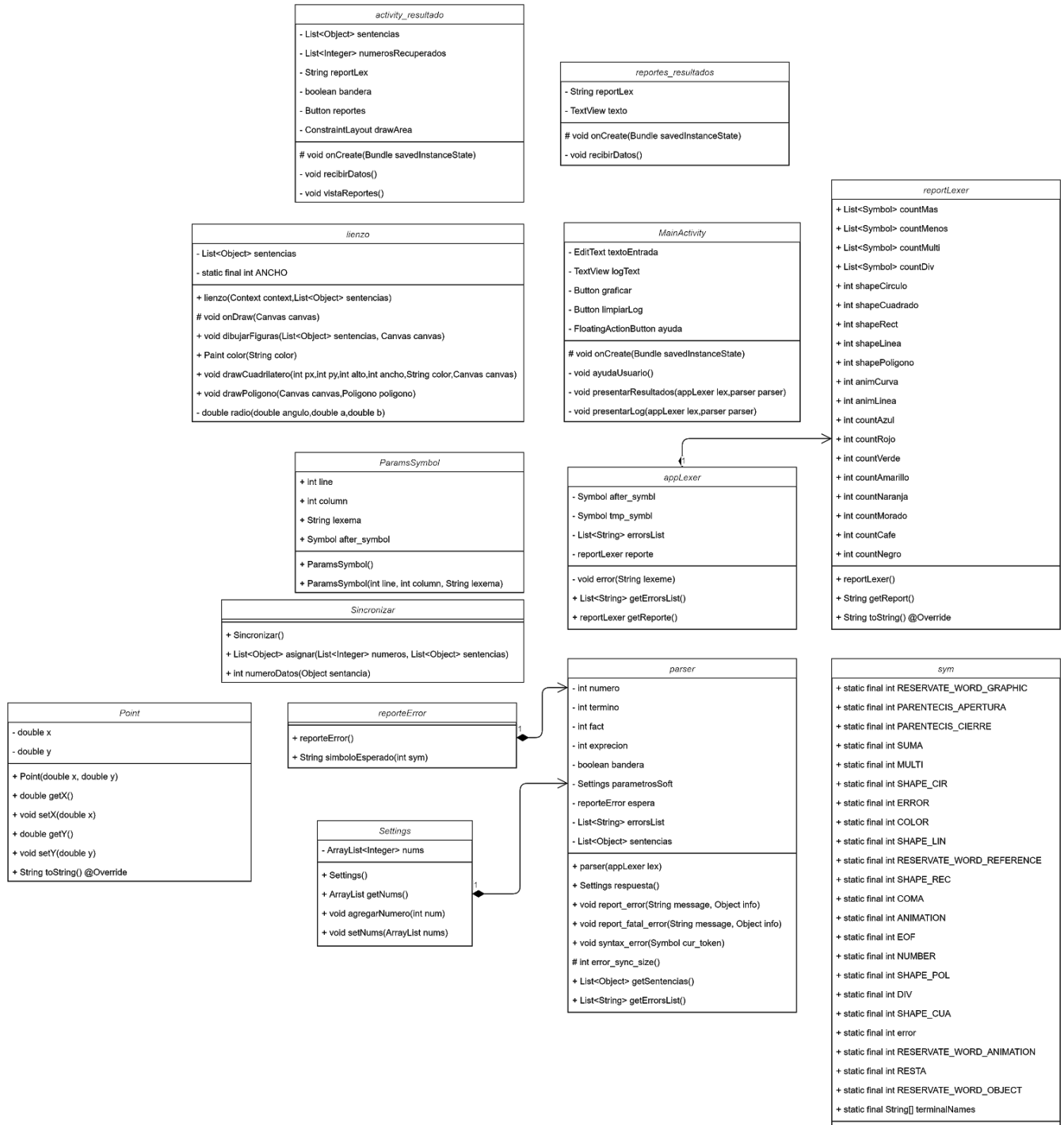
```



La producción anterior establece el conjunto primitivo, que puede ser un dígito o la agrupación de una expresión



## Diagrama de Clases



<i>Animar</i>
<ul style="list-style-type: none"> <li>- int destinox</li> <li>- int destinoy</li> <li>- string typeAnimation</li> </ul>
<ul style="list-style-type: none"> <li>+ Animar(int destinox,int destinoy,String typeAnimation)</li> <li>+ int getDestinox()</li> <li>+ void setDestinox(int destinox)</li> <li>+ int getDestinoi()</li> <li>+ void setDestinoi(int destinoy)</li> <li>+ String getTypeAnimation()</li> <li>+ void setTypeAnimation(String typeAnimation)</li> <li>+ String toString() @Override</li> </ul>

<i>Circulo</i>
<ul style="list-style-type: none"> <li>- int posx</li> <li>- int posy</li> <li>- int radio</li> <li>- String color</li> </ul>
<ul style="list-style-type: none"> <li>+ Circulo(int posx,int posy,int radio,String color)</li> <li>+ int getPosx()</li> <li>+ void setPosx(int posx)</li> <li>+ int getPosy()</li> <li>+ void setPosy(int posy)</li> <li>+ String getTypeAnimation()</li> <li>+ int getRadio()</li> <li>+ void setRadio(int radio)</li> <li>+ String getColor()</li> <li>+ void setColor(String color)</li> <li>+ String toString() @Override</li> </ul>

<i>Cuadrado</i>
<ul style="list-style-type: none"> <li>- int posx</li> <li>- int posy</li> <li>- int tamLado</li> <li>- String color</li> </ul>
<ul style="list-style-type: none"> <li>+ Cuadrado(int posx,int posy,int tamLado,String color)</li> <li>+ int getPosx()</li> <li>+ void setPosx(int posx)</li> <li>+ int getPosy()</li> <li>+ void setPosy(int posy)</li> <li>+ int getTamLado()</li> <li>+ void setTamLado(int tamLado)</li> <li>+ String getColor()</li> <li>+ void setColor(String color)</li> <li>+ String toString() @Override</li> </ul>

<i>Poligono</i>
<ul style="list-style-type: none"> <li>- int posx</li> <li>- int posy</li> <li>- int alto</li> <li>- int ancho</li> <li>- int lados</li> <li>- String color</li> </ul>
<ul style="list-style-type: none"> <li>+ Poligono(int posx,int posy,int alto,int ancho,int lados,String color)</li> <li>+ int getPosx()</li> <li>+ void setPosx(int posx)</li> <li>+ int getPosy()</li> <li>+ void setPosy(int posy)</li> <li>+ int getAlto()</li> <li>+ void setAlto(int alto)</li> <li>+ int getAncho()</li> <li>+ void setAncho(int ancho)</li> <li>+ int getLados()</li> <li>+ void setLados(int lados)</li> <li>+ String getColor()</li> <li>+ void setColor(String color)</li> <li>+ String toString() @Override</li> </ul>

<i>Rectangulo</i>
<ul style="list-style-type: none"> <li>- int posx</li> <li>- int posy</li> <li>- int alto</li> <li>- int ancho</li> <li>- String color</li> </ul>
<ul style="list-style-type: none"> <li>+ Rectangulo(int posx, int posy, int alto, int ancho, String color)</li> <li>+ int getPosx()</li> <li>+ void setPosx(int posx)</li> <li>+ int getPosy()</li> <li>+ void setPosy(int posy)</li> <li>+ int getAlto()</li> <li>+ void setAlto(int alto)</li> <li>+ int getAncho()</li> <li>+ void setAncho(int ancho)</li> <li>+ String getColor()</li> <li>+ void setColor(String color)</li> <li>+ String toString() @Override</li> </ul>


<i>Linea</i>
<ul style="list-style-type: none"> <li>- int posx</li> <li>- int posy</li> <li>- int posx2</li> <li>- int posy2</li> <li>- String color</li> </ul>
<ul style="list-style-type: none"> <li>+ Linea(int posx,int posy,int posx2,int posy2,String color)</li> <li>+ int getPosx()</li> <li>+ void setPosx(int posx)</li> <li>+ int getPosy()</li> <li>+ void setPosy(int posy)</li> <li>+ int getPosx2()</li> <li>+ void setPosx2(int posx2)</li> <li>+ int getPosy2()</li> <li>+ void setPosy2(int posy2)</li> <li>+ String getColor()</li> <li>+ void setColor(String color)</li> <li>+ String toString() @Override</li> </ul>

## Organización del Proyecto

El proyecto es administrado principalmente por las acciones realizadas por la pantalla principal (**MainActivity**) el cual recoge el texto obtenido y es enviado a la clase **appLexer** y a su vez es enviado a la clase **parser**, ambas clases trabajan en conjunto para poder descifrar por medio de las expresiones regulares y la gramática que anteriormente fue mostrada obtener los Tokens los cuales son guardados por medio del objeto **Symbol** del paquete **java-cup-runtime** en donde se empaqueta el tipo de Token y su token predecesor y a su vez para poder ir guardando otras características de cada token es utilizada la clase **ParamsSymbol**, en donde se almacenan la **línea**, **columna** y **lexema** de cada token encontrado, pasando a la clase **parser** esta es la encargada de ir descifrando las instrucciones de graficación y animación y la solución paso por paso de las operaciones matemáticas realizando las sumas, restas, multiplicaciones, divisiones y agrupaciones según la estructura dada en el texto en base a las reglas de producción, al momento de resolver la regla de producción de los números estos son guardados en una **lista de números recuperados** en la clase **Settings**; En **parser** de igual manera es donde se resuelven las figuras y las animaciones por hacer en el programa, asignándole en su clase correspondiente entre ellas **Circulo**, **Cuadrado**, **Rectangulo**, **Poligono**, **Linea**, **Animar** dichas clases empaquetan en un objeto el parametro de **color**; si durante la lectura hay algún error este es almacenado en una lista de la propia clase del **parser** o de **appLexer** al momento de detectar el error si es **lexico** se establece la recomendación de corrección y si es de tipo **sintactica** el **parser** mostrara una lista con los tokens que en realidad se esperaban, todo el error es almacenado en formato de cadena de caracteres, los resultados de errores son pedidos por el **MainActivity** y si la lista de errores ambas clases esta vacía entonces son enviados a sincronización de los números y sentencias recuperadas del proceso de compilación, si en el caso que no estén vacías alguna de ellas el **ActivityMain** mostrará los errores en la pantalla.

La lista de números encontrados y las sentencias recuperadas que son las figuras y animaciones, son enviadas al **activity\_resultado** en donde se pasa la sincronización de datos, la clase a utilizar es **Sincronización** el cual acomoda tanto los números y las instrucciones solucionadas por el analizador léxico y sintáctico.

La sincronización de los valores numéricos y sentencias tiene como resultado una lista de objetos que pueden ser **Circulo**, **Cuadrado**, **Rectangulo**, **Poligono**, **Linea**, **Animar**, y son enviados a una instancia de la clase **lienzo** la encargada de graficar todas las figuras, recibiendo el contexto de la actividad en ejecución. Dentro del clase **lienzo** la lista de objetos obtenida es tomada y se analiza cada objeto uno por uno y dentro de esa misma acción se procede a parsear el objeto y enviarlo al método el cual se encarga de dibujar, dimensionarlo y asignarle el color correspondiente; al momento de graficar un círculo y un



polígono en el programa se es necesario de la clase ***Point*** que almacena un punto coordinado para luego realizar la unión de los mismos y presentar la figura solicitada.

Por último ***reportes\_resultados*** el cual es una actividad muestra los resultados almacenados en ***reportLexer*** el cual contiene la cantidad de colores utilizados y tipos, figuras y tipos de animaciones, que son para la vista del usuario que a su vez es generado en la clase ***appLexer*** y que es enviado por las actividades intermediarias hasta esta instancia.