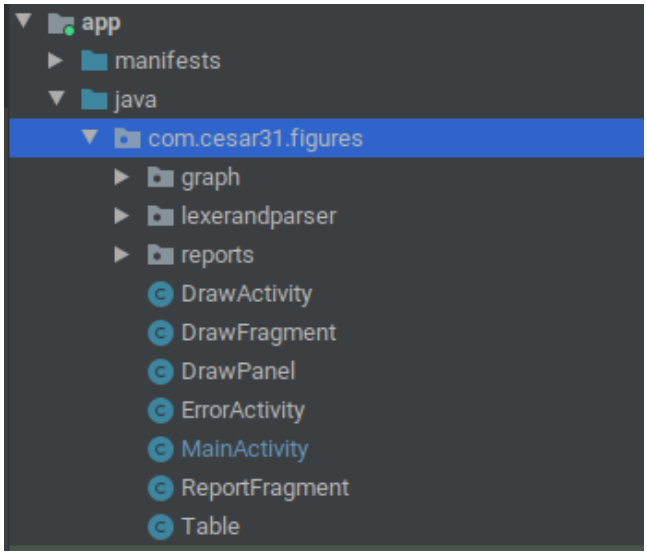
Manual Técnico

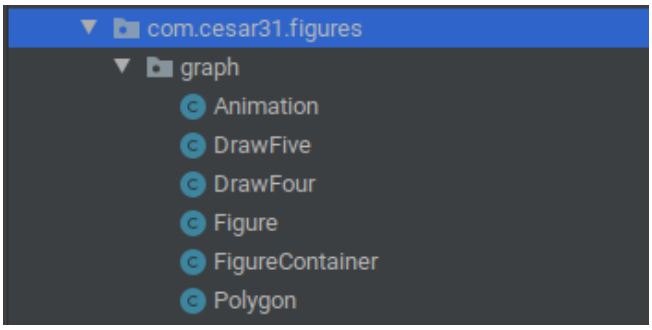
Organización del proyecto

El proyecto, organizado dentro del package `com.cesar31.figures` sigue la siguiente estructura.



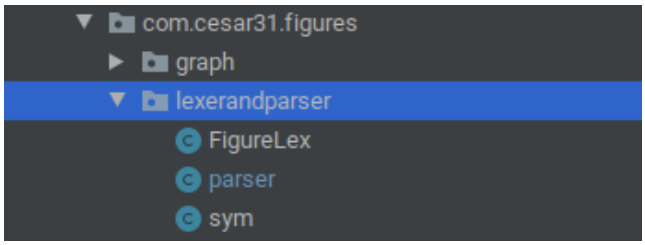
De manera general, las clases en el package `com.cesar31.figures` y fuera de los demás se utilizan para la lógica de la parte visual.

En el paquete `com.cesar31.figures.graph`

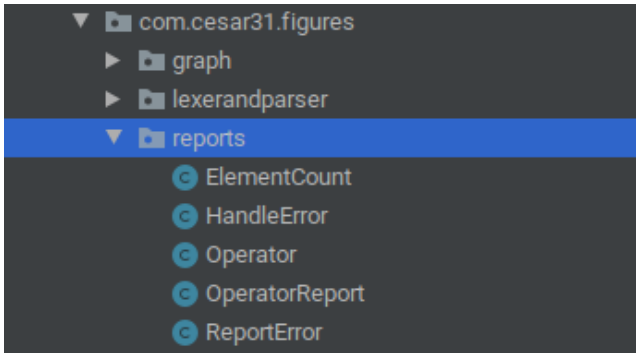


Se utiliza para almacenar las clases para almacenar los distintos objetos que se puedan crear para su posterior animación, la clase `Animation.java` se utiliza para guardar datos de animaciones. La clase `FigureContainer.java` se utiliza para almacenar listados de los distintos tipos de figuras, y también contiene una instancia de la clase `ElementCount` utilizada para generar los reportes de colores usados, animaciones hechas y figuras graficadas.

El paquete `com.cesar31.figures.lexerandparser` almacena las clases necesarias para el análisis léxico y sintáctico.



El paquete `com.cesar31.figures.reports` tiene las clases necesarias para generar los reportes de elementos utilizados(objetos, colores, animaciones) y también para generar reporte de errores.



Análisis de gramática para analizador léxico

La parte de análisis léxico, que esta a cargo de la clase `FigureLex.java` generada por la herramienta JFlex (https://jflex.de/). La descripción de las expresiones regulares es la siguiente:

```
Number = [0-9]+
LineTerminator = \r|\n|\r\n
```

```
WhiteSpace = {LineTerminator} | [\s\t\f]
```

Las expresiones regulares usadas para el desarrollo de la aplicación móvil que se muestran, se utilizan básicamente para obtener ocurrencias de números enteros, que juegan un papel importante para la realización de operaciones con los matches o coincidencias que esta expresión pueda devolver.

La expresión regular `LineTerminator` se utiliza para obtener las ocurrencias de saltos de línea en sus posibles combinaciones, esta se utiliza junto con `WhiteSpace` para poder ignorar todos los saltos de línea y espacios que se puedan encontrar.

También se usan las siguientes expresiones regulares que obtienes específicamente las coincidencias para todas las palabras reservas:

```
/* keywords */
"graficar"    { return symbol(GRAPH, yytext()); }
"circulo"   { return symbol(CIRCLE, yytext()); }
"cuadrado"    { return symbol(SQUARE, yytext()); }
"rectangulo"    { return symbol(RCTNGL, yytext()); }
"linea"     { return symbol(LINE, yytext()); }
"poligono"    { return symbol(POLYGN, yytext()); }
"animar"    { return symbol(ANIMT, yytext()); }
"objeto"    { return symbol(OBJ, yytext()); }
"anterior"    { return symbol(ANT, yytext()); }
"curva"     { return symbol(CURVE, yytext()); }

/* colors */
"azul"      { return symbol(COLOR, yytext()); }
"rojo"      { return symbol(COLOR, yytext()); }
"verde"     { return symbol(COLOR, yytext()); }
"amarillo"    { return symbol(COLOR, yytext()); }
"naranja"   { return symbol(COLOR, yytext()); }
"morado"    { return symbol(COLOR, yytext()); }
"cafe"      { return symbol(COLOR, yytext()); }
"negro"     { return symbol(COLOR, yytext()); }
```

Esta parte del archivo `figure.jflex` se encarga de obtener las palabras reservadas que encuentre en las diferentes entradas que el usuario pueda realizar.

```
/* signs and operators */

  ","     { return symbol(COMMA, yytext()); }
  "("     { return symbol(LPAREN, yytext()); }
  ")"     { return symbol(RPAREN, yytext()); }
  "+"     { return symbol(PLUS, yytext()); }
  "-"     { return symbol(MINUS, yytext()); }
```

```
  "*"     { return symbol(TIMES, yytext()); }
  "/"     { return symbol(DIV, yytext()); }
```

Aparte de las palabras reservadas, necesitamos encontrar todas las ocurrencias de los símbolos u operadores que se describen arriba, para posteriormente encontrar las posibles operaciones ingresadas por el usuario.

Posteriormente, necesitamos encontrar las ocurrencias de números y también espacios, esto último no interesa para algún tipo de operación en el análisis de la gramática, simplemente los encontramos para poder ignorarlos y no tomarlos como errores.

```
/* Numbers */
  {Number}    { return new Symbol(ENTERO, yyline + 1, yycolumn + 1, Integer.valueOf(yytext())); }

  /* whitespace */
  {WhiteSpace}    { /* Ignore */ }
```

Por último, todos los demás caracteres que nos encontramos y que no forman parte de lo anterior, se toman como error.

```
/* error */

[^]       { return symbol(ERROR, yytext()); }
```

Análisis de gramática para analizador sintáctico

La parte del análisis sintáctico esta a cargo de la clase, `parser.java` clase generada por la herramienta Java CUP (http://www2.cs.tum.edu/projects/cup/). Es importante señalar que la herramienta también genera la clase, `sym.java` que básicamente contiene un listado de los símbolos terminales de la gramática.

El archivo, `figure.cup` al compilarse gracias a Java CUP, genera las clases ya descritas. A continuación se brinda una breve descripción del archivo y de la gramática que se plasma en este.

En la parte de parser code, aparte del constructor se tiene lo siguiente:

```
private Symbol cur_token;
private boolean parsed;
private HandleError handleErrors;
private FigureContainer container;
```

La variable `parsed` se utiliza para determinar si el archivo fue parseado con éxito o no, luego los objetos `handleErrors` que se encarga de obtener los errores en las posibles entradas y `container` que se encarga de guardar los objetos que se van a graficar y también de construir los reportes de colores, objetos y animaciones.

Descripción de la gramática

Se tienen declarados los siguientes terminales y no terminales, también se hacen uso de terminales y no terminales de tipo Integer.

```
terminal GRAPH, CIRCLE, SQUARE, RCTNGL, LINE, POLYGN, ANIMT, OBJ, ANT, CURVE, COLOR;
terminal COMMA, LPAREN, RPAREN, PLUS, MINUS, TIMES, DIV, ERROR;
terminal Integer ENTERO;

non terminal expr, draw, anim, kanim, clr, opan, draw4, draw5, draw6;
non terminal Integer s, t, u, e;
```

La gramática tiene como símbolo inicial a, `expr` esta tiene las siguientes producciones

```
expr    ::=     draw anim expr
                | draw anim
                | draw expr
                | draw
                | error
                ;
```

Aquí `draw` se encarga de las producciones que leen instrucciones para graficar y `anim` de las producciones para revisar la estructura de instrucciones de animaciones. Luego la parte de `error`

para darle una "salida" a los posibles errores que se encuentre a este nivel de la gramática.

```
draw    ::=     GRAPH CIRCLE draw4:circle
                {:
                        container.setFour((DrawFour)circle, "circulo");
                        // System.out.println(" <- circulo");
                :}
                | GRAPH SQUARE draw4:square
                {:
                        container.setFour((DrawFour)square, "cuadrado");
                        // System.out.println(" <- cuadrado");
                :}
                | GRAPH RCTNGL draw5:rectangle
                {:
                        container.setFive((DrawFive) rectangle, "rectangulo");
                        // System.out.println(" <- rectangulo");
                :}
                | GRAPH LINE draw5:line
                {:
                        container.setFive((DrawFive) line, "linea");
                        // System.out.println(" <- linea");
                :}
                | GRAPH POLYGN draw6:polygon
                {:
                        container.setPolygon((Polygon) polygon);
                        // System.out.println(" <- poligono");
                :}
                | error draw4
                {: //System.out.println("draw ::= error draw4"); :}
                | error draw5
                {: //System.out.println("draw ::= error draw5"); :}
                | error draw6
                {: //System.out.println("draw ::= error draw6"); :}
                ;
```

Como se puede observar, el símbolo no terminal `draw` verifica las instrucciones para graficar círculos, cuadrados, rectángulos, líneas y polígonos, dentro de esta producción encontramos los no terminales `draw4`, `draw5`, `draw6`, se utilizan para verificar la estructura de los parámetros para graficar, circulos y cuadrados (`draw4`), rectángulos y líneas (`draw5`) y polígonos (`draw6`).

También es importante señalar que las producciones `error draw4`, `error draw5` y `error draw6` se utilizan para dar una salida en caso de tener algún error en momento de escribir la sentencia graficar figura, y dirigir la producción hacia los parámetros de la figura en busca de posibles errores.

Como ya se mencionó, `draw4`, `draw5` y `draw6` se usan para verificar la estructura de los parámetros de las figuras a graficar.

```
draw4   ::=     LPAREN s:x COMMA s:y COMMA s:r COMMA clr:color RPAREN
                {:
                        RESULT = new DrawFour(x, y, r, (String)color);
                        // System.out.printf("draw4(%d, %d, %d, %s)", x, y, r, color);
                :}
                ;

draw5   ::=     LPAREN s:x COMMA s:y COMMA s:h COMMA s:w COMMA clr:color RPAREN
                {:
                        RESULT = new DrawFive(x, y, h, w, (String)color);
                        // System.out.printf("draw5(%d, %d, %d, %d, %s)", x, y, h, w, color);
                :}
                ;

draw6   ::=     LPAREN s:x COMMA s:y COMMA s:h COMMA s:w COMMA s:n COMMA clr:color RPAREN
                {:
                        RESULT = new Polygon(x, y, h, w, n, (String) color);
                        // System.out.printf("draw6(%d, %d, %d, %d, %d, %s)", x, y, h, w, n, color);
                :}
                ;
```

Luego las producciones `anim`, `opan`, `clr` y `kanim` se utlizan para revisar la estructura de instrucciones para animación, parámetros de animaciones, colores y tipos de animación respectivamente.

```
anim    ::=     ANIMT OBJ ANT opan:animation
                {:
                        container.setAnimation((Animation) animation);
                       //  System.out.println(" <- animar objeto anterior");
                :}
                | error opan
                {: //System.out.println("anim ::= error opan"); :}
                ;

opan    ::=     LPAREN s:xf COMMA s:yf COMMA kanim:kind RPAREN
                {:
                        RESULT = new Animation(xf, yf, (String)kind);
                        // System.out.printf("opan(%d, %d, %s)", xf, yf, kind);
                :}
                ;

clr     ::=     COLOR:color
                {: RESULT = color; :}
                | error
                {: //System.out.println("clr ::= error"); :}
                ;

kanim   ::=     CURVE:t1
                {: RESULT = t1; :}
                | LINE:t2
                {: RESULT = t2; :}
```

```
                | error
                {: //System.out.println("kanim ::= error"); :}
```

Y por último, las producciones `s`, `t`, `u` y `e` todos no terminales enteros se encargan de revisar que las operaciones que el usuario ingrese, tengan sentido. Se observa que esta gramática respeta la precedencia de operadores matemáticos.

```
s       ::=     s:n1 PLUS t:n2
                {: RESULT = n1 + n2; :}
                | s:n1 MINUS t:n2
                {: RESULT = n1 - n2; :}
                | t:n1
                {: RESULT = n1; :}
                | error
                {: //System.out.println("s ::= error"); :}
                ;

t       ::=     t:n1 TIMES u:n2
                {: RESULT = n1 * n2; :}
                | t:n1 DIV u:n2
                {: RESULT = n1 / n2; :}
                | u:n1
                {: RESULT = n1; :}
                ;

u       ::=     MINUS e:n1
                {: RESULT =  -n1; :}
                | e:n1
                {: RESULT = n1; :}
                ;

e       ::=     ENTERO:n1
                {: RESULT = n1; :}
                | LPAREN s:n1 RPAREN
                {: RESULT = n1; :}
                ;
```