**Universidad de San Carlos de Guatemala**
**Facultad de Ingeniería**
**Escuela de Ciencias y Sistemas**
**Organización de Lenguajes y Compiladores 2**
**Segundo Semestre 2016**
**Catedráticos:** Ing. Bayron López, Ing. Edgar Sabán
**Tutores académicos:** Julio Flores, Andrea Alvarez, Jordy González, Mario Asencio

# Lienzo 2D

Primer proyecto de laboratorio

# 1. Tabla de contenidos

## Contenido

# 2. Objetivos

## 2.1. Objetivo General

Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 2 en la creación de soluciones de software.

## 2.2 Objetivos Específicos

- Aplicar los conceptos de compiladores para implementar el proceso de interpretación de código de alto nivel.
- Aplicar los conceptos de compiladores para analizar un lenguaje basado en los principios de la Programación Orientada a Objetos.
- Implementar un módulo de depuración de software al momento de realizar el análisis de código de alto nivel.

# 3. Descripción general del proyecto

Para el primer proyecto de laboratorio se requiere la elaboración de una herramienta que interpretará un código de programación llamado "Lenguaje de lienzos" y de esta forma será capaz de generar imágenes de dos dimensiones sobre un lienzo de dibujo. Esta herramienta recibirá el nombre de “Lienzo 2D”.

La aplicación contará con un entorno de desarrollo, el cual contará con editor y un área gráfica de lienzos 2D, en la cual se mostrarán las imágenes creadas durante la interpretación del código. Esta área gráfica se mostrará en una ventana independiente al editor para adaptarse al tamaño de la imagen a mostrar.
El editor contará con varias funcionalidades que harán más cómoda la programación de imágenes para el usuario, entre ellas están: crear y modificar archivos, numeración de líneas, información de la posición en línea y columna del cursor y múltiples pestañas de edición en paralelo.
Además de estas características el editor deberá contener una funcionalidad de depurador integrado, el cual debe permitir al usuario interpretar su código de manera controlada, cambiando la velocidad de ejecución línea por línea y observando la creación de la imagen en tiempo real.
El lenguaje que deberá ser interpretado se basa en el paradigma de programación orientado a objetos, por lo que incluye características determinantes de este paradigma: abstracción en clases, polimorfismo y sobrecarga de métodos, herencia entre objetos, modularidad, encapsulamiento y ocultación. Mediante la herencia, se permitirá unir las distintas imágenes generadas por cada clase del lenguaje, para combinar todas las capas de los mismos y generar una sola imagen.

Finalmente, se requiere la elaboración de un servidor web, llamado Galería de Arte, en el cual podrán ser publicadas las imágenes que el usuario genere mediante la aplicación. Para la exportación hacia la galería se requiere que al finalizar la interpretación, en el área gráfica 2D del editor exista una funcionalidad de publicador, el cual convertirá la información de la imagen a un formato JSON y la enviará mediante red a la Galería de Arte.
Las imágenes que se hayan exportado hacia la Galería de Arte deberán estar disponibles en cualquier momento para su visualización.

A continuación una descripción gráfica del flujo de la aplicación, incluyendo la depuración, interpretación y generación de la imagen, publicación hacia la Galería de Arte y visualización desde un navegador web:

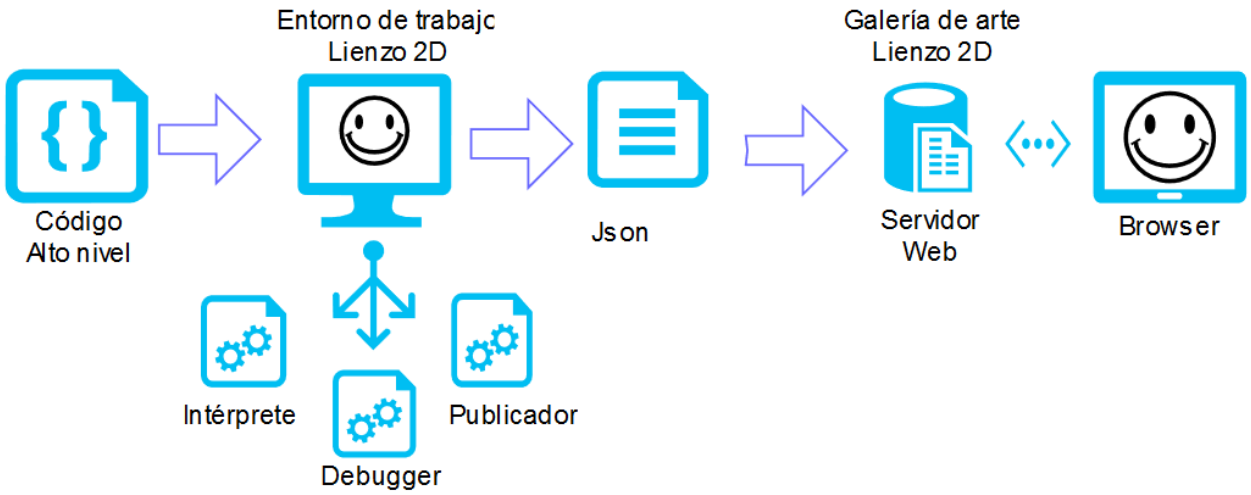



Figura 1: Diagrama de flujo de la aplicación

La aplicación Lienzo 2D deberá tener los siguientes módulos:

## 3.1 Intérprete

El entorno de trabajo tendrá un intérprete de código de alto nivel que permitirá la generación de imágenes. El intérprete ejecutará las instrucciones del código de alto nivel (ver sección 7) y generará la tabla de símbolos correspondiente, o el reporte de errores en caso que existan. Al finalizar la interpretación deberá mostrar la imagen resultante dentro de la aplicación en el área de gráficos 2D.
El intérprete deberá ser capaz de llevar a cabo todas las instrucciones del lenguaje.

## 3.2 Depurador

El debugger o depurador será un módulo del entorno de trabajo que habilitará la opción de ejecución controlada del código fuente. El debugger deberá indicar qué línea de código se está realizando en la generación del lienzo, para ello deberá marcarla y mostrará el resultado en el área gráfica del Lienzo en tiempo real.
El depurador deberá funcionar de manera automática; es decir, ejecutará las instrucciones de forma continua. La velocidad en la que se ejecutan las instrucciones podrá definirse por el usuario.

En la sección 5 se detalla el funcionamiento del depurador.

## 3.3 Publicador de lienzo

El módulo publicador, permitirá la comunicación del entorno de trabajo y el servidor web Galería de Arte.
La comunicación entre el entorno de trabajo y la Galería de Arte, deberá realizarse por medio de socket, servlets o apache thrift (queda a discreción del estudiante utilizar cualquiera de las 3 opciones dadas).

A continuación se describe el formato de envío de la información a  la galería de arte:

```
{
   "<Nombre del Lienzo>": {
      "Comentario": "<Comentario>",
      "Imagen": "<Array de bits>",
         "Fecha": "<dd/mm/aaaa hh:mm:ss>",
         }
}
```

Ejemplo:


```
{
   "Arbol_ceiba": {
      "Comentario": "Lienzo que representa la ceiba",
      "Imagen":
"000100111101110101000111101010101010101010100000
00000000000000000001010101010010101010011111100011
1111101011111111111111111",
         "Fecha": "18/08/2016 20:00:00",
         }
}
```


A continuación se describen las funcionalidades del editor y área de gráficos del Entorno de trabajo, los cuales permiten la integración de los módulos descritos anteriormente, y la galería de arte de Lienzo 2D:

# 4. Entorno de trabajo “Lienzo 2D”

El entorno de trabajo estará compuesto por un editor y un área gráfica y será el principal medio de comunicación entre la aplicación y usuario. En el editor se ingresará el código fuente (Lenguaje de lienzos, ver sección 7), y en el área gráfica de Lienzo, se mostrará el resultado de la ejecucion del codigo fuente.

## 4.1 Editor

El editor será parte del entorno de trabajo, cuya finalidad será proporcionar ciertas funcionalidades, características y herramientas que serán de utilidad al usuario. La función principal del editor será el ingreso el código fuente (Lenguaje de lienzos) que será interpretado. El editor deberá contar con lo siguiente:

### 4.1.1 Funcionalidades

- **Crear**: El editor deberá ser capaz de crear archivos en blanco.
- **Abrir archivos**: El editor deberá abrir archivos en formato .lz
- **Guardar el archivo**: El editor deberá guardar el estado del archivo en el que se estará trabajando.
- **Guardar el archivo como:** el editor deberá guardar el archivo en el que se estará trabajando con la extensión y ruta que el usuario desee.

### 4.1.2 Características

- **Múltiple pestañas:** el editor deberá ser capaz de crear todas las pestañas que el usuario desee.
- **Enumeración líneas**: el editor deberá tener un contador de líneas de código ingresado.
- **Información de posición del cursor:** el editor deberá mostrar numéricamente la línea y columna en la que se encuentra el cursor en todo momento.

### 4.1.3 Herramientas

- **Ejecutar**: Invocará al intérprete.
- **Debuggear código:** Invocará al depurador.
- **Velocidad del debugger**: este una característica del debugger el cual servirá para variar la velocidad en que se ejecutan las líneas de código.

### 4.1.4 Reporte de errores

La aplicación deberá poder generar reporte de errores, será una herramienta que permitirá la identificación de los errores léxicos, sintácticos y semánticos, en el momento de interpretar Lenguaje de alto nivel que ayuda a dibujar Lienzos. Estos errores deben ser almacenados en un archivo en formato html, para más detalle del formato del reporte ver sección 9.

### 4.1.5 Tabla de símbolos

La aplicación deberá poder generar la tabla de símbolos correspondientes al análisis del código de alto nivel. Dentro del menú de herramientas se encontrará la opción de tabla de símbolos, la cual debe mostrar la tabla de todos símbolos de la última ejecución que se realizó en la herramienta en la sección 8 se encuentra detallado el formato que se requiere, para mostrar la tabla.

Para la interfaz del editor se sugiere la siguiente:

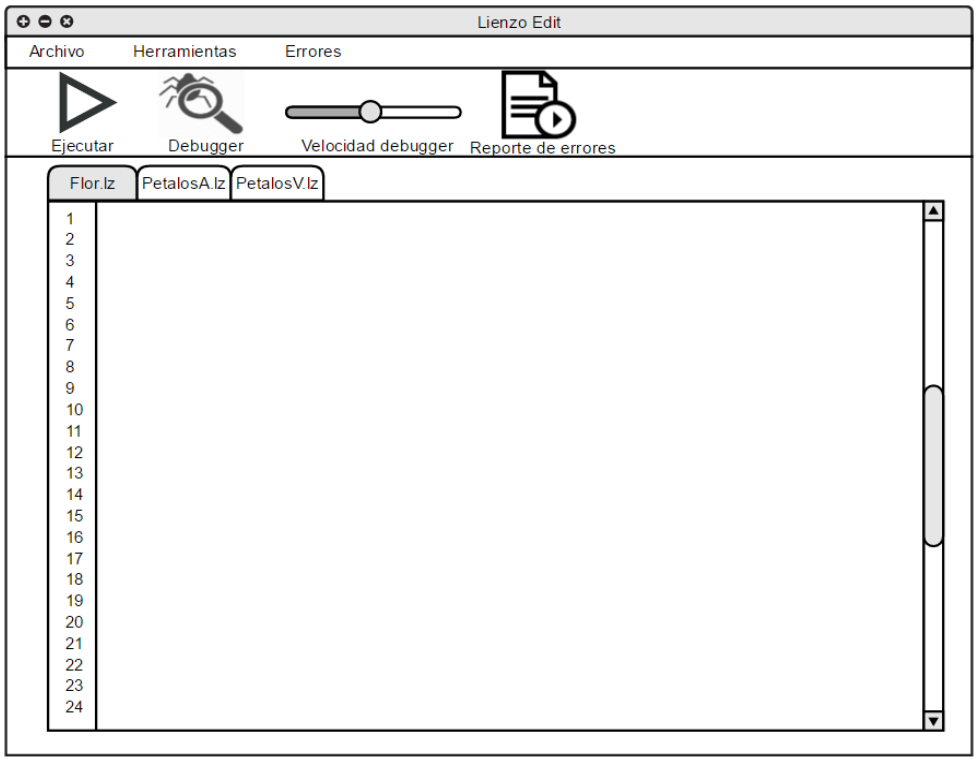



Figura 3: Interfaz sugerida del editor

## 4.2 Área gráfica del Lienzo

El área gráfica del lienzo será la parte en blanco de una ventana donde se mostrará figuras, colores y caracteres creados con el lenguaje de lienzos (definida en la sección No. 7). Se podrá acceder a una parte en específico del área gráfica del lienzo mediante coordenadas X & Y por lo que el área gráfica del lienzo deberá tener obligatoriamente reglas guías (medidas en pixeles) en los extremos. El área gráfica debe implementarse en una ventana distinta al editor.

En la interfaz del área gráfica, deberá tener un botón que permite publicar lienzos en la galería de arte, para la publicación de lienzo se requiere colocar el nombre y una breve descripción del lienzo. A continuación se muestra la interfaz sugerida para el área gráfica:

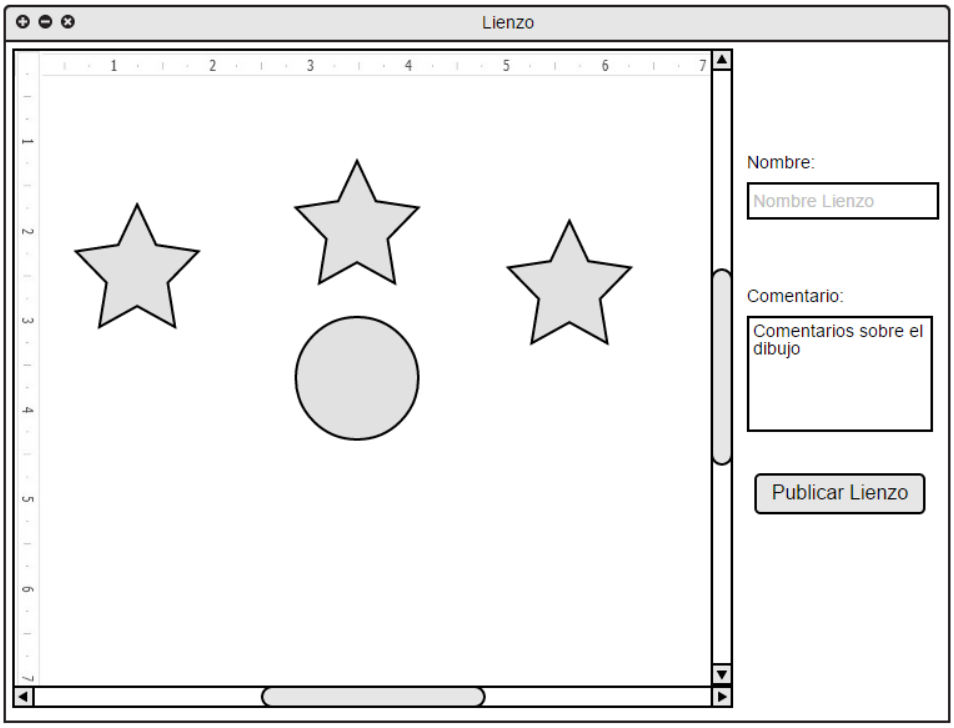



Figura 4: Interfaz sugerida para el área gráfica del lienzo

# 5. Depurador

El debugger o depurador será un módulo del entorno de trabajo ligado al editor, desde el cual podrá invocarse para ejecutar de manera controlada el código fuente.

Para indicar qué instrucción se está ejecutando en todo momento, el depurador la marcará resaltando la línea con color amarillo en el editor y mostrará el resultado en el área gráfica del Lienzo en tiempo real, como se muestra en el siguiente ejemplo (Las flechas negras únicamente son para referencia del ejemplo. Estas no deben graficarse):

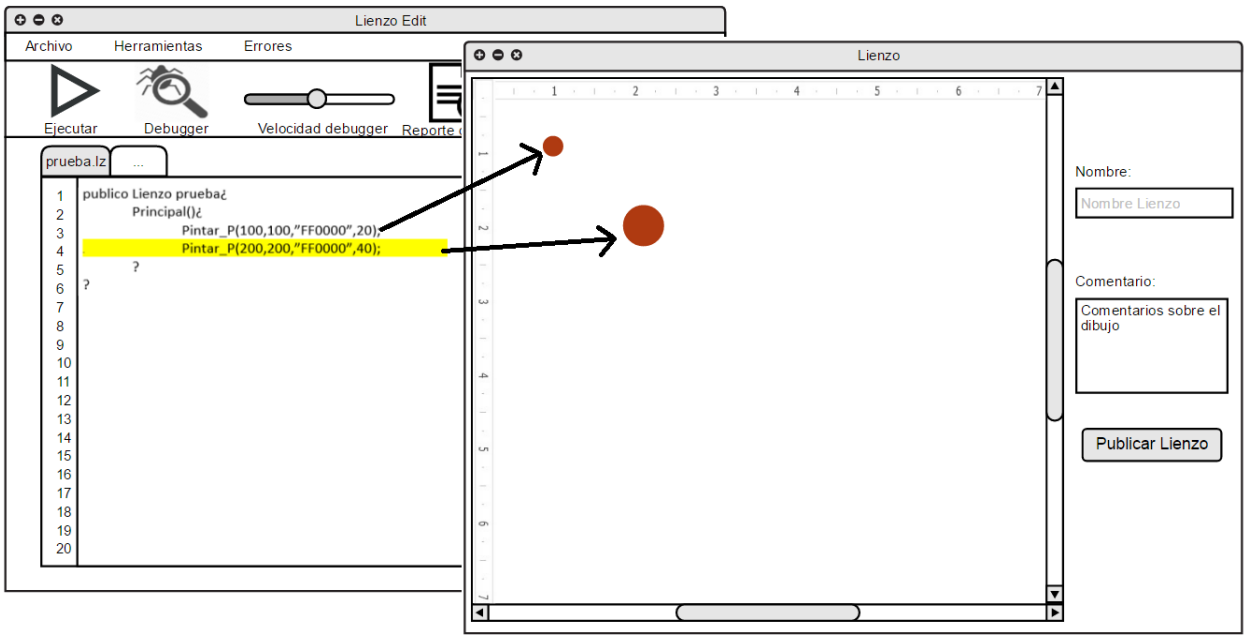
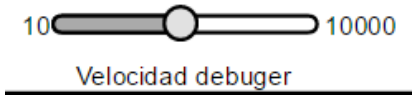



Figura 2: Interfaz sugerida para el debugger

## Modalidad del depurador:

La modalidad de ejecución del depurador será automática, por lo que el funcionamiento que deberá presentar es el siguiente: después de iniciarse, debe ejecutar las instrucciones de manera continua, a una velocidad que será determinada por el usuario. Al presionar la tecla “F3” del teclado el depurador deberá pausar la ejecución y podrá reanudarse presionando la tecla “F3” de nuevo. Al presionar la tecla “F4” se cancelará la depuración.

En el módulo del depurador automático se tendrá la capacidad de poder configurar la velocidad en que el intérprete ejecuta el código fuente. La velocidad podrá ser cambiada en cualquier momento mediante un Slider colocado en el editor con una escala de 10 a 10,000 milisegundos por instrucción, es decir el debugger puede esperar de una escala de 10 a 10,000 milisegundos antes de ejecutar la siguiente instrucción.




*Slider de 10 a 10,000 milisegundos*

# 6. Galería de arte “Lienzo 2D”

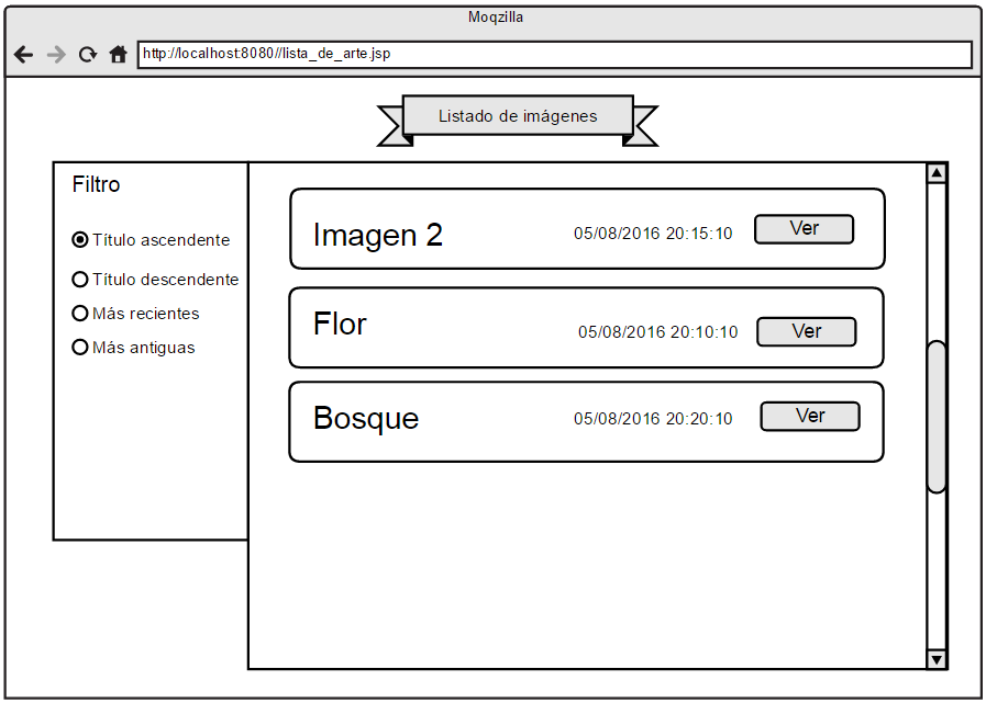
La galería de arte será un servidor web que contendrá todos los lienzos que han sido publicados. El servidor de la galería deberá ser desarrollado en JSP, el cual contará con un canal de comunicación (socket, servlets o apache thrift) que escuchará la publicación de lienzos.

La galería recibirá una cadena en formato JSON, y deberá de extraer la información recibida para procesarla y almacenarla de la forma que el estudiante considere más adecuada.

Dentro de la Galería de Arte existirán 2 páginas web:

- /lista_de_arte.jsp:
  Dentro de esta página se mostrará una lista incluyendo el título y la fecha de publicación de todas las imágenes almacenadas en la galería. Estas debe poder ordenarse ya sea por orden alfabético del título de manera ascendente o descendente o en base al orden de publicación ya sean las más recientes o las más antiguas primero.
  Al hacer click sobre una de las imágenes de la lista deberá redirigirse hacia la página de /galeria.jsp y mostrarse la imagen en sí.
- /galeria.jsp:
  Dentro de esta página se mostrará la imagen seleccionada, incluyendo su título, fecha de publicación y su comentario si es que tiene uno. Además se mostrarán enlaces a los lados que permitirán navegar por las imágenes.

A continuación se muestra un ejemplo de la página lista_de_arte:



A continuación se muestra un ejemplo de la página galeria, al haber seleccionado la imagen “Flor de primavera”:

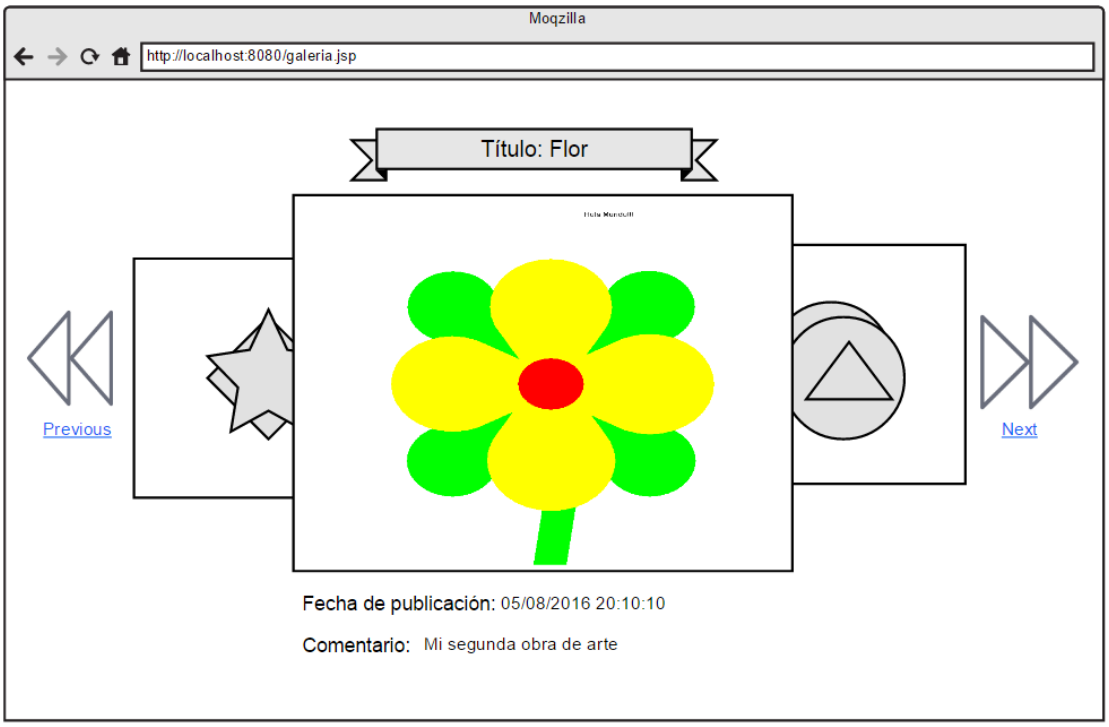



Figura 5: Página sugerida para la galería de arte

A continuación se describe el lenguaje orientado a objetos a utilizar:

# 7. Descripción del Lenguaje de lienzos

El lenguaje de lienzos es un lenguaje de programación basado en el paradigma de Programación Orientada a Objetos, donde un lienzo tomará el papel de un objeto o clase y mediante sus instrucciones, ciclos, bifurcaciones, etc. podrá generar un dibujo en 2 dimensiones. A continuación se describe el lenguaje de lienzos.

## 7.1 Sensibilidad ante mayúsculas

El lenguaje será capaz de reconocer diferencia entre mayúsculas y minúsculas para todas las palabras que acepta el lenguaje por lo que si se declara un atributo de nombre “Contador” este será distinto a definir un atributo “contador”. Para que no haya problema con las palabras reservadas, estas solo pueden venir en minúsculas o la primera letra en mayúsculas y todas las demás en minúsculas.

## 7.2 Comentarios

- Comentario de una línea: Estos deben empezar con dos signos de “mayor que” y terminar con un salto de línea:

  >>comentario de una sola línea

- Comentario multilínea: Estos deben empezar con un signo de “menor que” y un guión y termina con un guión y un signo de “mayor que”:

  <- comentario de varias líneas Segunda línea Tercera línea …. ->

## 7.3 Carácter de finalización y encapsulamiento de sentencias

Toda sentencia terminará siempre con el símbolo de dólar $ y el cuerpo de todo lienzo deberá de encapsularse por signos de interrogación ¿?. Al igual que en los ciclos esta encapsulación debe de mantenerse, si esto no se cumple se tomará como un error sintáctico

Ejemplo:

```
NombreLienzo ¿
        //Sentencia 1 $
        //Sentencia 2 $
        //Ciclo 1 ¿
                //Sentencia 1$
                //Sentencia 2 $
                //Ciclo 1_1 ¿
                        //Sentencia 1 $
                        //Sentencia 2 ….. $
                ?
                //Sentencia 3 …. $
        ?
        //Sentencia 3 $
?
```

## 7.4 Tipos de dato

Los tipos de dato que el lenguaje deberá soportar en el valor de una variable son los siguientes:

| Nombre | Descripción | Ejemplo | Observaciones |
|---|---|---|---|
| Entero | Este tipo de dato acepta solamente números enteros | 1, 50, 100, 255125, etc. | No tiene límite de tamaño |
| Doble | Es un entero con decimal | 1.2, 50.23, 00.34, etc. | Se manejará como regla general el manejo de 6 decimales |
| boolean | Admite valores de verdadero o falso, y variantes de los mismos | verdadero, falso, true, false. 1=verdadero, 0=falso | si se asigna los enteros 1 o 0 debe aceptar como verdadero o falso según el ejemplo |
| carácter | Solamente admite un carácter por lo que no será posible ingresar cadenas enteras. Viene encerrado en comilla simple entre a-z, A-Z, -, _, : incluyendo símbolos y caracteres de escape #t, #n | ‘a’, ’b’, ’c’, ’E’, ‘Z’’, ‘1’, ‘2’, ‘^t’, ‘^n’, ‘^’, ‘%’, ‘)’, ‘=’, ‘!’, ‘&’, ‘/’, ‘\’, etc | En el caso de querer escribir comilla simple se escribirá primero ^ y después la comilla simple ‘^’’, además si se quiere escribir ^ se pondrá 2 veces ‘^^’, los caracteres de escape se escriben siempre ‘^t’ y ‘^n’ |
| cadena | Este es un conjunto de | “cadena1” | En el caso de querer |

| | | | |
|---|---|---|---|
| | caracteres que pueden ir separados por espacios en blanco encerrados en comilla doble, los posibles caracteres a ingresar serán los mismos que los del tipo caracter solo que agregando a estos el espacio | “est#$%o es una ca&/(de=na #n#n!!!” | escribir comilla doble se escribirá primero # y después la comilla doble “ho#”la mundo #””, además si se quiere escribir # se pondrá 2 veces"##", por ejemplo “ho#la mundo ##”, los caracteres de escape se escriben siempre #t y #n |

## 7.5 Operadores Relacionales

Son los símbolos utilizados en las expresiones, su finalidad es comparar expresiones entre sí dando como resultado booleanos. En el lenguaje serán soportados los siguientes

| Operador | Descripción | Ejemplo |
|---|---|---|
| == | Igualación: Compara ambos valores y verifica si son iguales<br>● iguales=true no<br>● iguales=false | 1==1, “hola”==”hola”, 25.2555==80.005 |
| != | Diferenciación: Compara ambos lados y verifica si son distintos<br>● distintos=true<br>● iguales=false | 1!=2, martes!=vari1 |
| < | Menor que: Compara ambos lados y verifica si el derecho es mayor que el izquierdo<br>● derecho mayor=true | (5/(5+5))<(8*8) |

| | | |
|---|---|---|
| | • derecho no mayor =false | |
| <= | Menor o igual que: Compara ambos lados y verifica si el derecho es mayor o igual que el izquierdo<br>• derecho mayor o igual=true<br>• derecho no mayor o igual =false | 55+66<=44 |
| > | Mayor que: Compara ambos lados y verifica si el izquierdo es mayor que el derecho<br>• izquierdo mayor=true<br>• izquierdo no mayor =false | (5+5.5)>8.98 |
| >= | Mayor o igual que: Compara ambos lados y verifica si el izquierdo es mayor o igual que el derecho<br>• izquierdo mayor o igual=true<br>• izquierdo no mayor o igual =false | 5-6>=4+6 |
| !&¡ | Nulo: Verifica si la variable fue declarada pero no le fue asignado algún valor inicial o una asignación posterior | (!&¡cad1) |

## 7.6 Operadores Lógicos

Símbolos para poder realizar comparaciones a nivel lógico de tipo falso y verdadero, sobre expresiones.

| Operador | Descripción | Ejemplo | Observaciones |
|---|---|---|---|
| \|\| | OR: Compara expresiones lógicas y si al menos una es verdadera entonces devuelve=verdadero en otro caso retorna=falso | (55.5<4) \|\| (!¡bandera)<br>resultado =verdadero | bandera=nulo |
| && | AND: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve =verdadero en otro caso retorna =falso | (flag1) && (“hola” == ”hola”)<br>resultado=falso | flag1=falso |
| !&& | NAND: Compara expresiones lógicas y si son ambas verdaderas entonces devuelve =falso en otro caso retorna =verdadero | (flag2) !&& (“hola”==”hola”)<br>resultado=falso | flag2=verdadero |
| !\|\| | NOR: Compara expresiones lógicas y si al menos una es verdadera entonces devuelve=falso en otro caso retorna=verdadero | (band1) !|| (!¡bandera)<br>resultado =verdadero | band1=falso<br>bandera= no nulo |

| | | | |
|---|---|---|---|
| &\| | XOR: Compara expresiones lógicas si al menos una es verdadera, pero no ambas entonces devuelve=verdadero en otro caso retorna=falso | (44+6!=4) &\| (1+2>60)<br><br>resultado =verdadero | |
| ! | NOT: Devuelve el valor inverso de una expresión lógica si esta es verdadera entonces devolverá = falso, de lo contrario retorna =verdadero | !(!¡ais)<br><br>resultado=falso | ais=nulo |

Precedencia de análisis y operación de las expresiones lógicas:

| Nivel | Operador | Asociatividad |
|---|---|---|
| 0 | NOT | Derecha |
| 1 | AND, NAND | Izquierda |
| 2 | OR, NOR, XOR | Izquierda |

Nota: 0 es el nivel de mayor importancia

## 7.7 Operadores Aritméticos

Símbolos para poder realizar operaciones de tipo aritmética sobre las expresiones en las que se incluya estos mismos.

### 7.7.1 Suma

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más **+**

Especificaciones sobre la suma:

- Al sumar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- Al sumar dos datos de tipo carácter (char, string) el resultado será la concatenación de ambos datos.

- Al sumar un dato numérico con un dato de tipo carácter el resultado será la concatenación de del dato de tipo carácter y la conversión a cadena del dato numérico.
- Al sumar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso utilizaremos la suma como la operación lógica or.

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| int + double<br>double + int<br>double + char<br>char + double<br>bool + double<br>double + bool<br>double + double | double | 5+4.5 = 9.5<br>7.8+3 = 10.8<br>15.3 + 'a' = 112.3<br>'b' + 2.7 = 100.7<br>true + 1.2 = 2.2<br>4.5 + false = 4.5<br>3.56+2.3 = 5.86 |
| int + char<br>char + int<br>bool + int<br>int + bool<br>int + int | int | 7 + 'c' = 106<br>'C' + 7 = 74<br>4 + true = 5<br>4 + false = 4<br>4 + 5 = 9 |
| string + int<br>string + double<br>double + string<br>int + string | string | "hola" + 2 = "hola2"<br>"hola"+3.5 = "hola3.5"<br>4.5 + "hola" = "4.5hola"<br>8+"hola" = "8hola" |
| string + char<br>char + string<br>string + string | string | "hola"+'t' = "holat"<br>'u'+"hola" = "uhola"<br>"hola"+ "mundo" = "holamundo" |
| bool + bool | bool | 1 + false = true = 1<br>true + true = true = 1<br>0 + 1 = 1 = true<br>false + 0 = false = 0 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.7.2 Resta:

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos **-**

Especificaciones sobre la resta:

- Al restar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible restar datos numéricos con tipos de datos de tipo caracter (string)
- No es posible restar tipos de datos caracter (char, string) entre sí.
- No es posible restar tipos de datos lógicos (bool) entre sí.

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| int - double<br>double - int<br>double - char<br>char - double<br>bool - double<br>double - bool<br>double - double | double | 5-4.5 = 0.5<br>7.8-3 = 4.8<br>15.3 - ‘a’ = -81.7<br>‘b’ - 2.7 = 95.3<br>true - 1.2 = -0.2<br>4.5 - false = 4.5<br>3.56 - 2.3 = 1.26 |
| int - char<br>char - int<br>bool - int<br>int - bool<br>int - int | int | 7 - ‘c’ = -92<br>‘C’ - 7 = 60<br>4 - true = 3<br>4 - false = 4<br>4 - 5 = -1 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.7.3 Multiplicación:

Operación aritmética que operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco *****

Especificaciones sobre la multiplicación:

- Al multiplicar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible multiplicar datos numéricos con tipos de datos caracter (string)
- No es posible multiplicar tipos de datos caracter (char, string) entre sí.
- Al multiplicar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso usaremos la multiplicación como la operación AND entre ambos datos.

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| int * double<br>double * int<br>double * char<br>char * double<br>bool * double<br>double * bool<br>double * double | double | 5*4.5 = 22.5<br>7.8*3 = 23.4<br>15.3 * ‘a’ = 1484.1<br>‘b’ * 2.7 = 264.6<br>true * 1.2 = 1.2<br>4.5 * false = 0<br>3.56 * 2.3 = 8.188 |
| int * char<br>char * int<br>bool * int<br>int * bool<br>int * int | int | 7 * ‘c’ = 693<br>‘C’ * 7 = 469<br>4 * true = 4<br>4 * false = 0<br>4 * 5 = 20 |

| | | |
|---|---|---|
| bool * bool | bool | true * 0 = 0 = false<br>true * 1 = 1 = true<br>0 * false = 0 = false<br>1 * false = 0 = false |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.7.4 División:

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal **/**

Especificaciones sobre la división:

- Al dividir dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible dividir datos numéricos con tipos de datos de tipo caracter (string)
- No es posible dividir tipos de datos caracter (char, string) entre sí.
- No es posible dividir tipos de datos lógicos (bool) entre sí.
- Al dividir un dato numérico entre 0 deberá arrojarse un error de ejecución

| **Operandos** | **Tipo de dato resultante** | **Ejemplos** |
|---|---|---|
| int / double<br>double / int<br>double / char<br>char / double<br>bool / double<br>double / bool<br>double / double<br>int / char<br>char / int<br>bool / int<br>int / bool<br>int / int | double | 5/4.5 = 1.11111<br>7.8/3 = 2.6<br>15.3 / ‘a’ =0.1577<br>‘b’ / 2.7 = 28.8889<br>true / 1.2 = 0.8333<br>4.5 / false = error<br>3.56 / 2.3 = 1.5478<br>7 / ‘c’ = 0.7070<br>‘C’ / 7 = 9.5714<br>4 / true = 4.0<br>4 / false = error<br>4 / 5 = 0.8 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.7.5 Potencia:

Operación aritmética que consiste en multiplicar varias veces un mismo factor. El operador de la potencia es el acento circunflejo **^**

Especificaciones sobre la potencia:

- Al potenciar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible potenciar datos numéricos con tipos de datos de tipo caracter (string)
- No es posible potenciar tipos de datos caracter (char, string) entre sí.
- No es posible potenciar tipos de datos lógicos (bool) entre sí.

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| int ^ double<br>double ^ int<br>double ^ char<br>char ^ double<br>bool ^ double<br>double ^ bool<br>double ^ double | double | 5^4.5 = 1397.54<br>7.8^3 = 474.55<br>15.3 ^ ‘a’ = <<fuera de rango>><br>‘b’ ^ 2.7 = 237853.96<br>true ^ 1.2 = 1.0<br>4.5 ^ false = 1.0<br>3.56 ^ -2.3 = 0.0539 |
| int ^ char<br>char ^ int<br>bool ^ int<br>int ^ bool<br>int ^ int | int | 7 ^ ‘c’ = <<fuera de rango>><br>‘C’ ^ 7 = 6060711605323<br>4 ^ true = 4<br>4 ^ false = 1<br>4 ^ 5 = 1024 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

Precedencia de análisis y operación de las expresiones lógicas:

| Nivel | Operador |
|---|---|
| 0 | Potencia |
| 1 | división, multiplicación |
| 2 | Suma, Resta |

Nota: 0 es el nivel de mayor importancia

## 7.8 Lienzo

La sintaxis para la declaración de un lienzo es la siguiente, en donde debe de establecerse el nombre del lienzo:

```
Lienzo Nombre¿
        //cuerpo del lienzo ….. $
?
```

Ejemplo:

```
Lienzo Arte ¿
        //cuerpo del lienzo $
?
```

## 7.9 Extender un Lienzo (Herencia)

Cada lienzo tendrá la posibilidad de poder extender atributos, métodos y funciones de otro lienzo indicando el nombre del lienzo de la cual se desea obtener toda su estructura, esto permitirá que se logre la unión de varios lienzos para formar uno solo. Al momento de extender un lienzo, se deberá buscar el lienzo padre en el archivo binario de lienzos cargados y compiladas para poder extender el lienzo hijo (función similar al “import” tradicional). La extensión de un lienzo tiene las siguientes características:

1. El método principal de ejecución de lienzos no será heredado de un lienzo a otro lienzo, por lo que el lienzo a extender puede tener su método principal de ejecución y el lienzo que extiende puede no tener este método. Al momento de dibujar dicho lienzo dará error por no tener método principal de ejecución dado a que este no existe o no fue definido.
2. Se puede definir la extensión de más de un lienzo, aunque esto puede traer el problema de que estos lienzos posean métodos o funciones con el mismo nombre y que al extender no se pueda saber cuál tomar, la solución a esto es que la última función, método o variable declarada será la que permanezca al final si y solo si esta tiene la propiedad “Conservar”, por lo que si el lienzo el cual se está definiendo tiene un método el cual fue heredado previamente de otro lienzo, este sustituirá al método heredado solo si tiene la propiedad “Conservar”, sino quedará el método, función o variable del último lienzo extendido.

Extender de otro lienzo es opcional para cada lienzo que se vaya a crear.

```
Lienzo Nombre [Extiende1, Extiende2,.... ExtiendeN] ¿
        //cuerpo del lienzo ….. $
?
```

Ejemplo:

```
Lienzo Arbol extiende Hojas, Arte, etc ¿
         //Sentencia 1 $
        //Sentencia 2 $
        //Ciclo 1 ¿
                //Sentencia 1 …..$
        ?
?
```

## 7.10 Visibilidad

Las visibilidades o modificadores de acceso de los objetos serán la forma manejar encapsulamiento dentro del proyecto. Esto quiere decir que será la forma de controlar el acceso a los datos que conforman un objeto.

### 7.10.1 público:

Este tipo de visibilidad es el más permisivo de todos. Cualquier componente perteneciente a un objeto **público** podrá ser accedido desde cualquier objeto o instancia.

```
publico Lienzo Fondo ¿
        //Cuerpo del lienzo $
?
```

### 7.10.2 privado:

Este tipo de visibilidad es el más restrictivo de todos. Cualquier componente perteneciente a un objeto **privado** solo podrá ser accedido por ese mismo objeto y nada más.

```
privado Lienzo Cuerpo ¿
        //Cuerpo del lienzo $
?
```

### 7.10.3 protegido:

Este tipo de visibilidad permitirá el acceso a los componentes pertenecientes a un objeto protegido únicamente desde el mismo objeto u objetos que hereden de él.

```
protegido Lienzo Cielo ¿
        //Cuerpo del lienzo $
?
```

## 7.11 Declaración y Asignación de Variables

La declaración puede hacerse desde cualquier parte del código ingresado, pudiendo declararlas desde ciclos, métodos, funciones o fuera de estos siempre dentro de un Lienzo. La forma normal en que se declaran las variables es la siguiente:

### 7.11.1 Declaración:

Lo que está encerrado en corchetes es opcional puede no venir.

Estructura:

**[Conservar] var Tipo nombre[,nombre2,nombre3…. , nombre" n"] [ Asignación ] $**

Ejemplo:

```
var entero numero1$
Conservar var cadena cad1,cad2,cad3,ejex,ejey $
```

En este caso las variables no tienen valor alguno pues se declaró pero no se le ha asignado valor, además cad1, cad2, cad3, ejex y ejey son tipo cadena.

```
Conservar var doble contador = 50.5, contador2 = 30.55$
var boolean f2=false,f1=true $
var cadena palabra2,palabra3 = "esto es un ejemplo :D #n"+"v" $
var boolean bandera2,bandera3=true $
Conservar var boolean flag2,flag3=!¡contador $
var boolean flag4=flag2 $
Conservar var caracter letra="A" $
```

Para las variables declaradas en un mismo lugar separadas por coma y que se les asigna un valor, a todas se les asigna el mismo valor.

### 7.11.2 Asignación:

Se deberá validar que la variable exista y que el valor sea del tipo correcto  Nombre Asignación

Ejemplo:

```
numero1 = 25$   //entero
entero numero1=true$   //entero
boolean contador = 50.5*20+115*b $
```

Con la asignación el casteo correspondiente para que la integridad del tipo de dato de la variable se mantenga, se debe realizar de acuerdo con la siguiente tabla:

| Tipo Variable | Tipo Valor | Resultado de casteo |
|---|---|---|
| entero | entero | entero |
| entero | cadena | error |
| entero | boolean | entero |
| entero | doble | entero |
| entero | carácter | entero ascii |
| cadena | <cualquier tipo> | cadena |
| boolean | <cualquier tipo> | error |

| | | |
|---|---|---|
| doble | entero | doble |
| doble | doble | doble |
| doble | cadena | error |
| doble | boolean | error |
| carácter | entero | ascii |
| carácter | carácter | carácter |
| carácter | <cualquier tipo> | error |

## 7.12 Asignación de símbolos de operaciones simplificadas

Con la finalidad de poder realizar la simplificación de asignaciones de valores y el trabajo sobre ellas, el lenguaje podrá soportar operadores de operaciones simplificadas

Aparte del uso mostrado para los operadores ++ y -- se podrán usar sobre las variables para modificar su valor en expresiones según la tabla:

### 7.12.1 Aumento:

Operación aritmética que consiste en añadir una unidad a un dato numérico. El aumento es una operación de un solo operando. El aumento sólo podrá venir del lado derecho de un dato. El operador del aumento es el doble signo más **++**

Especificaciones sobre el aumento:

- Al aumentar un tipo de dato numérico (int, double, char) el resultado será numérico.
- No es posible aumentar tipos de datos caracter (string).
- No es posible aumentar tipos de datos lógicos (bool).
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc)

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| double++ | double | 4.56++ = 4.57 |
| int ++<br>char ++ | int | 15++ = 16<br>'a'++ = 98 = 'b' |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.12.2 Decremento:

Operación aritmética que consiste en quitar una unidad a un dato numérico. El decremento es una operación de un solo operando. El decremento sólo podrá venir del lado derecho de un dato. El operador del decremento es el doble signo menos **--**

Especificaciones sobre el decremento:

- Al decrementar un tipo de dato numérico (int, double, char) el resultado será numérico.
- No es posible decrementar tipos de datos caracter (string).
- No es posible decrementar tipos de datos lógicos (bool).
- El decremento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc)

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| double-- | double | 4.56-- = 3.57 |
| int --<br>char -- | int | 15-- = 14<br>'b'-- = 97 = 'a' |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

### 7.12.3 Suma simplificada:

Realiza una suma con la variable que está al lado izquierdo y la expresión que está del lado derecho. El operador de la suma simplificada es signo más y signo igual **+=**

Especificaciones sobre el aumento:

- Al sumar simplificadamente un tipo de dato numérico (int, double, char) el resultado será numérico.
- Concatenará tipos de datos caracter (string).
- No es posible sumar simplificadamente tipos de datos lógicos (bool).
- La suma simplificada podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc)

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| Char += char<br>Char += string<br>Char += int<br>Char += double<br>string+= string<br>string+= char<br>string += int<br>string += double | string | 'a'+='b' = "ab"<br>"abc"+= 'd' = "abcd"<br>'a'+=1 = "a1"<br>"abc"+= 2.5 = "abc2.5" |
| int += int<br>int += double<br>int += char | int | 15+= 1 = 16<br>15+= 2.5 = 17.5<br>1+= 'a' = 98 |
| double += double<br>double += int<br>double+= char | double | 4.56+= 1.50 = 6.06<br>5.2+=1=6.2<br>1.5+='a' =98.5 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica

### 7.12.4 Resta simplificada:

Realiza una resta con la variable que está al lado izquierdo y la expresión que está del lado derecho. El operador de la resta simplificada es signo menos y signo igual **-=**

Especificaciones sobre el aumento:

- Al restar simplificadamente un tipo de dato numérico (int, double, char) el resultado será numérico.
- Eliminará caracteres tipos de datos string.
- No es posible restar simplificadamente tipos de datos lógicos (bool).
- La resta simplificada podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc)

| Operandos | Tipo de dato resultante | Ejemplos |
|---|---|---|
| string -= int | string | "ab"-=1 = 'a'<br>"abc"-= 1 = "ab" |
| int -= int | int | 15-= 1 = 14 |

| | | |
|---|---|---|
| int -= double<br>int -= char | | 15-= 2.5 = 12.5<br>100-= ‘a’ = 3 |
| double -= double<br>double -= int<br>double-= char | double | 4.56-= 1.50 = 3.06<br>5.2-=1=4.2<br>100.5+=’a’ =3.5 |

Cualquier otra combinación es inválida y deberá arrojar un error de semántica

## 7.13 Declaración de arreglos

Se pueden realizar declaraciones de arreglos dentro del lenguaje de la aplicación, por lo que la manera de declaración y asignación es parecida a la de variables y es la siguiente:

**[Conservar] var Tipo arreglo <nombre> [,nombre2,nombre3…. , nombre” n”] <dimensiones> [ = Asignación ] $**

Nota: Lo que está encerrado entre corchetes es opcional, y lo que está dentro de los tags es arbitrario

Las dimensiones será de manera “N” dimensionales, por lo que no se limita al arreglo a un número específico de dimensiones el mínimo serán de 1 dimensión.

Estructura:

**[<expresión>][<expresión>].....[<expresión>]**

Ejemplo:

```
var entero arreglo arr1 [a+b+(a+(5+1)%5)]$    //con 1 dimensión
Conservar var entero arreglo arr2,arr3,arr4  [5][5*5+5-4]$   //con 2 dimensiones
var entero arreglo arr5[arr1[1-1]][(arr1[5+arr2[0][4]]/5)+1][1][8][15]$ //con 5 dimensiones
```

También permitirá asignar valores al arreglo cuando se declare, si no se asigna un valor todas las casillas comienzan con valor nulo o que no tienen valor asignado. Para poder asignar al momento de inicializar se utilizara las llaves “{“”}” para indicar los arreglos para cada dimensión y se separa con coma por cada posición o casilla del arreglo.

1 dimensión {<expresion>,<expresion>, ……,<expresion>}
2 dimensiones {{<expresion>,...,<expresion>},{<expresion>,…,<expresion>}}

n dimensiones
{{<expresion>,...,<expresion>},{<expresion>,…,<expresion>},...,{<expresion>,...,<expresion>}}

Ejemplo:

var entero arreglo arr0 [3]={5,10,15}$ //En este caso arr0 tiene en la primera posición 5,10 en la segunda y 15 en la tercera.

var entero arreglo arr10,arr20,arr30 [2][3]={{5,10,15},{20,25,30}}$ // en este caso arr10, arr20 y arr30 poseen una matriz de 2*3 por lo que en cada posición del arreglo de 2 tiene 3 elementos.

**Nota: lo que puede ir en cada casilla para definir su valor es una expresión.)**

## 7.14 Asignación de arreglos

Una vez definido el arreglo se puede asignar valor a cada posición del arreglo por lo que solamente se debe especificar la posición:

Estructura:

**<Nombre> <Dimensiones> = <expresión>**

Ejemplo:

```
arr0[2] = 5+5$    // al arreglo 0 en la posición 2 se le asigna el valor 10
arr1 [(a+(5+1)%5)] = arr0[2]+5*8+b $
arr4 [5+arr1[5]] [5*arr1[4]+5*arr1[5]-4] = arr1[(a+5)%2] $
```

**Nota: Si el resultado de la expresión excede el tamaño de la dimensión, deberá mostrarse como un error semántico.**

## 7.15 Sentencia “Si”

Esta sentencia comprobará la condición y ejecutará su contenido en caso de que esta sea verdadera. La estructura de una sentencia “si” es la siguiente:

```
si(condición)¿
        // Sentencia 1  $
        // Sentencia 2  $
        // Sentencia 3  $
        …
```

```
        // Sentencia n  $
?
```

Ejemplo:

```
si((a+b)>(2+var1))¿
        // Sentencia 1  $
        // Sentencia 2  $
        …
        // Sentencia n  $
?
```

También puede utilizarse la palabra “sino” para ejecutar otro contenido en caso de que la condición sea falsa, de la siguiente manera:

```
si(condición)¿
        // Sentencia 1  $
        // Sentencia 2 $
         // Sentencia 3 $
        …
        // Sentencia n $
?sino¿
        // Sentencia 1  $
        // Sentencia 2  $
        // Sentencia 3  $
        …
        // Sentencia n $
?
```

Ejemplo:

```
si((b>a)||(b>c))¿
        // Sentencia 1  $
        // Sentencia 2 $
         // Sentencia 3 $
```

```
        …
        // Sentencia n $
?sino¿
        // Sentencia 1  $
        // Sentencia 2  $
        // Sentencia 3  $
        …
        // Sentencia n $
?
```

## 7.16 Sentencia “comprobar”

Esta sentencia de control evaluará una variable y ejecutará un contenido dependiendo del valor de ésta, así como también podrá establecerse un contenido “defecto” (opcional en la sentencia) por si la variable no contiene ningún valor de los establecidos previamente. La estructura de una sentencia de control “comprobar” es la siguiente:

```
comprobar(variable) ¿
        caso valor1:
                // Sentencias caso 1   $
                [salir]  $
        caso valor 2:
                 // Sentencias caso 2   $
                [salir]  $
        caso valor 3:
                // Sentencias caso 3  $
                [salir]$
         …
        defecto:
                // Sentencias default $
                [salir]  $
?
```

Ejemplo:

```
comprobar(var1) ¿
        caso valor1:
```

```
                // Sentencias caso 1   $
                salir $
        caso valor 2:
                 // Sentencias caso 2   $
        caso valor 3:
                // Sentencias caso 3  $
        defecto:
                // Sentencias default $
                salir  $
?
```

Aquí se introduce una nueva palabra reservada, “salir”, por medio de la cual se podrá salir de cualquier ciclo o sentencia de control sin ejecutar el código que se encuentre por debajo de esta palabra.

## 7.17 Sentencia “Para”

Este ciclo debe tener una asignación, seguido de una comparación o una expresión booleana y por último una sentencia de incrementación o disminución en unidades. El ciclo ejecutará su contenido hasta que la condición o expresión booleana sea verdadera por lo que si la condición inicialmente es verdadera el ciclo no se ejecuta. Los signos de incrementos o disminución son “++” y “--” respectivamente.

La estructura de un ciclo “para” es la siguiente:

```
para(asignación; condición; incremento|disminución/acción posterior)¿
        // Sentencia 1  $
        // Sentencia 2 $
        // Sentencia 3  $
        …
        // Sentencia n  $
?
```

**Nota: La variable de asignación puede ser declarada antes o inicializada directamente en la misma asignación, así como la acción posterior debe de haber cualquier acción posterior desde una asignación con incremento o disminución u otra acción**

Ejemplo:

```
var entero i$
        para(i=0; i<10; i++)¿
                // Sentencia 1  $
                // Sentencia 2 $
                // Sentencia 3  $
        ?
```

## 7.18 Sentencia “Mientras”

Este ciclo ejecutará su contenido siempre que se cumpla la condición que se le dé por lo que podría no ejecutarse si la condición es falsa desde el inicio.  La estructura de un ciclo “mientras” es la siguiente:

```
mientras(condición)¿
        // Sentencia 1$
        // Sentencia 2$
        // Sentencia 3$
        …
        // Sentencia n $
?
```

Ejemplo:

```
mientras(a<=b)¿
        a++$
        //sentencias $
?
```

## 7.19 Sentencia “Hacer-Mientras”

Este ciclo ejecutará al menos 1 vez su contenido, luego comprobará la condición para determinar si debe o no ejecutarse nuevamente.  La estructura de un ciclo “Hacer-Mientras” es la siguiente:

```
hacer¿
        // Sentencia 1$
        // Sentencia 2$
        // Sentencia 3$
        …
        // Sentencia n$
?mientras(condición)$
```

Ejemplo:

```
hacer¿
        // Sentencia 1$
        // Sentencia 2$
        // Sentencia 3$
        …
        // Sentencia n$
?mientras((1+2)<=(a+b+c)&&(a==b))$
```

## 7.20 Sentencia “Continuar”

Para los ciclos puede venir la palabra reservada “Continuar” que lo que hace es que ignora las demás instrucciones siguientes a partir de donde fue escrita de la iteración en curso y pasa a la siguiente iteración.

Ejemplo:

```
hacer¿
        // Sentencia 1$
        // Sentencia 2$
        Continuar$
        // Sentencia 3$
?mientras((1+2)<=(a+b+c)&&(a==b))$
```

Por lo que en este se ejecutará las sentencias 1 y 2, pero siempre se obviaba la sentencia 3 por lo que nunca se ejecutaría, en todas las iteraciones del ciclo while.

**Nota: Para el uso de las sentencias salir o continuar se debe verificar el ambiente, si el ambiente no es el correcto deberá marcar error.**

## 7.21 Funciones y Procedimientos

Las funciones y procedimientos sólo pueden ser llamados desde dentro de algún procedimiento o función definida dentro del lienzo. La declaración de funciones y procedimientos se realizará de la siguiente manera:

```
[Conservar ] [Tipo] nombre([tipo parametro1, tipo parametro2, . . .tipo parametro n])¿
        // Sentencia 1
        // Sentencia 2
        …
        // Sentencia n
        [retorna valor]
?
```

En esta parte se introduce la palabra reservada “retorna” para indicar el valor (o variable) a retornar en el método. El tipo es opcional, esto quiere decir que en caso de no ser escrito, el método se tomará como un “void” por lo que no será necesario establecer un valor de retorno. El prefijo Conservar establece su obligatoriedad al declararse/sobrescribirse

Ejemplo:

```
Int funcion1( int parametro1, string parametro2)¿
                var entero numero2$
                numero2+=parametro1$
                retorna numero2$
?
```

### Sobrecarga

Varias funciones pueden poseer el mismo nombre, la forma de distinguir estos será por medio del número y tipo de parámetros que requieran en su declaración.

Ejemplo:

```
Int funcion1( int parametro1, string parametro2)¿
                    var entero numero2$
                    numero2+=parametro1$
                   retorna numero2$
?

Int funcion1(char parametro1)¿
                    var entero numero2$
                    numero2+=parametro1$
                   retorna numero2$
?
```

## 7.22 Funciones y Procedimientos Nativos del Lenguaje

Las funciones y procedimientos sólo pueden ser llamados desde dentro de algún procedimiento o función definida dentro del lienzo.

### 7.22.1 Pintar punto

Este método tomará una serie de parámetros con los cuales podrá dibujar un punto en una posición en específica del área gráfica del lienzo su sintaxis es la siguiente:

```
Pintar_P(posición en X, posición en Y, color, Diámetro)$
```

- El primer parámetro deberá ser de tipo entero he indicará la posición en X del área gráfica del lienzo donde se pintara el punto.
- El segundo parámetro deberá ser de tipo entero he indicará la posición en Y del área gráfica del lienzo donde se pintara el punto.
- El tercer parámetro deberá ser de tipo cadena he indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero he indicará el diámetro medido en pixeles de punto.

*Ejemplo:*

```
Pintar_P(100,100, “#000000”, 50)$
```

### 7.22.2 Pintar óvalo/rectángulo

Este método tomará una serie de parámetros con los cuales podrá dibujar un óvalo es decir con altura y anchura con tamaños distintos en una posición en específica del área gráfica del lienzo su sintaxis es la siguiente:

```
Pintar_OR(posición en X, posición en Y, color, Ancho, Alto, figura)$
```

- El primer parámetro deberá ser de tipo entero he indicará la posición en X del área gráfica del lienzo donde se pintara el óvalo o rectángulo.
- El segundo parámetro deberá ser de tipo entero he indicará la posición en Y del área gráfica del lienzo donde se pintara el óvalo o rectángulo.
- El tercer parámetro deberá ser de tipo cadena he indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.
- El cuarto parámetro deberá ser de tipo entero he indicará el ancho medido en pixeles del óvalo o rectángulo.
- El quinto parámetro deberá ser de tipo entero he indicará el alto medido en pixeles del óvalo o rectángulo.
- El sexto parámetro deberá ser de tipo carácter y sólo podrá tomar los siguientes valores ‘o’ para dibujar un óvalo y ‘r’ para dibujar un rectángulo.

*Ejemplo:*

```
Pintar_OR(100,100, “#000000”, 50, 100,’o’)$ //dibuja un óvalo
Pintar_OR(100,100, “#000000”, 50, 100,’r’)$ //dibuja un rectángulo
```

### 7.22.3 Pintar cadena

Este método tomará una serie de parámetros con los cuales podrá dibujar una cadena en una posición en específica del área gráfica del lienzo su sintaxis es la siguiente:

```
Pintar_S(posición en X, posición en Y, color, Cadena)$
```

- El primer parámetro deberá ser de tipo entero he indicará la posición en X del área gráfica del lienzo donde se pintara la cadena.
- El segundo parámetro deberá ser de tipo entero he indicará la posición en Y del área gráfica del lienzo donde se pintara la cadena.
- El tercer parámetro deberá ser de tipo cadena he indicará el color del punto. dicho parámetro será ingresado en hexadecimal, ejemplo “#FFFFFF”, “#FFCCEE”, etc.

- El cuarto parámetro deberá ser de tipo cadena y será el texto a desplegar en el área gráfica del lienzo

  *Ejemplo:*

```
Pintar_S(100,100, “#000000”, “Hola mundo”)$
```

### 7.22.4 Método Principal del Lienzo

Éste método es el que el intérprete buscará al momento de dibujar el lienzo, a partir de este método el usuario puede llamar a otros métodos o funciones en donde define las sentencias necesarias para poder crear un Lienzo que utilice como base otros lienzos, con el fin de formar uno solo.
Si un lienzo no tiene método principal no tendrá como dibujar y por lo tanto no deberá ser desplegado, aunque es permitido definir un Lienzo sin método principal dado a que esta lienzo puede que solo sirva para extender otros lienzos.

La forma en que este método se declara es la siguiente:

```
Principal()¿
        // Sentencia 1$
        // Sentencia 2$
        ……
?
```

Como se ha mencionado anteriormente el lenguaje no soporta la extensión/herencia de métodos principales entre Lienzos.

### 7.22.5 Ordenar:

El método ordenar recibe como parámetro un arreglo y un parámetro para indicar de qué manera se quiere ordenar, si el ordenamiento tuvo éxito retorna un valor de 1, si la forma de ordenar no es aceptada o no se realizó el ordenamiento retorna 0.

Estructura:

```
Ordenar (arreglo  <arreglo a ordenar>,tipo <forma de ordenar>)
```

Retorna:

entero

Dónde:

- arreglo a ordenar: es una variable tipo arreglo que puede ser de tipo entero, cadena, caracter o doble
- forma de ordenar: es la manera de como ordena el arreglo. Se guiará como la siguiente tabla

| Forma de ordenar | Acción | Aplica |
|---|---|---|
| ascendente | ordena de menor a mayor | todos |
| descendente | ordena de mayor a menor | todos |
| pares | ordena los pares primero y después todos los demás | entero, doble(tomando solo la parte entera), carácter (según orden ascii) |
| impares | ordena los impar primero y después todos los demás | entero, doble(tomando solo la parte entera), carácter (según orden ascii) |
| primos | ordena los primos primero y después todos los demás | entero, doble(tomando solo la parte entera), carácter (según orden ascii) |

### 7.22.6 Sumarizar:

El método sumarizar extrae la suma de todos los elementos del arreglo para lo que ese será su salida, recibe como parámetro un arreglo, su retorno es una cadena con la suma de todos sus elementos.

Estructura:

Sumarizar (arreglo <arreglo a sumar>)

Retorno:

cadena

Dónde:

- arreglo a sumar: es una expresión tipo arreglo que puede ser de tipo entero, cadena, caracter o doble

Ejemplo:

```
var entero arreglo arr0 [3]={5,10,15} $
var caracter arreglo arr1 [4]={„a“,“b“,“F“,“5“} $
var doble arreglo arr3 [2]={1.5,2.6} $
var cadena cad $

cad=Sumarizar(arr0)$     // valor de cad: “30”
cad=Sumarizar(arr1)$    // valor de cad: “abF5”
cad=Sumarizar(arr3)$     // valor de cad: “4.1”
cad=Sumarizar({“hola”,” ”,”mundo”})$   // valor de cad: “hola mundo”
```

# 8. Tabla de Símbolos

La tabla de símbolos es un reporte que se mostrará en el editor y contiene una estructura de datos donde mostrará cada símbolo perteneciente al lenguaje con información relevante de este.

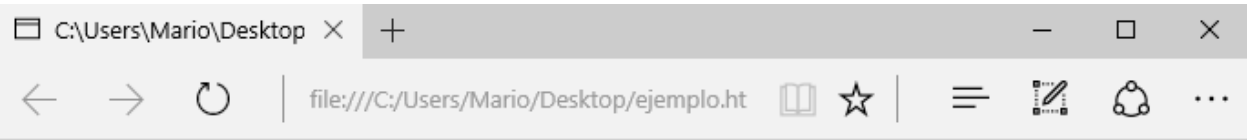
La opción se encontrará en el menú de herramientas del editor, y esta mostrará la tabla de símbolos de la última ejecución realizada por la herramienta.

La tabla de símbolos se mostrará como un reporte, del programa interpretado, en formato html. Este reporte debe contener el día y hora de la ejecución, seguido de la tabla que contendrá el detalle de los símbolos relevantes del programa de entrada. La tabla de símbolos deberá contener como mínimo la siguiente información: nombre o identificador del símbolo, tipo de dato del símbolo, rol (si es un atributo, función parámetro, etc) y ámbito.

A continuación se muestra un ejemplo con la tabla de símbolos que se debe desplegar:

```
public Lienzo PaisajeFondo¿
    Conservar publico entero[] Circulos(entero posx, entero posy, entero radio)¿
        var int arreglo arr[2] = {posx,posy}$
        para(posx=100;posx<500;posx+=1)¿
            Paint_P(posx+50, posy--, “#d11919”, radio*2)$
        ?
    ?
    retorna arr$
?
```

Tabla de símbolos de salida:




# Tabla de símbolos

**Día de ejecución: 13 de Agosto de 2016**

**Hora de ejecución: 12:30:15 p.m.**

| Id | Tipo | Rol | Ambito |
|---|---|---|---|
| PaisajeFondo | Lienzo | Lienzo | Principal |
| Circulos | entero | metodo | PasisajeFondo |
| posx | entero | atributo | PaisajeFondo |
| posy | entero | atributo | PaisajeFondo |
| radio | entero | atributo | PaisajeFondo |
| arr | arreglo de enteros | atributo | PaisajeFondo |

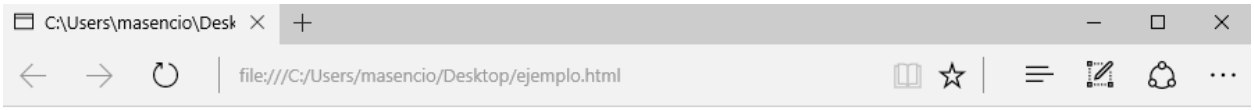
# 9. Manejo de Errores

## Reporte de errores

La herramienta deberá ser capaz de identificar todo tipo de errores (léxicos, sintácticos y semánticos) al momento de interpretar lenguaje de entrada (Lenguaje Lienzo). Estos errores se almacenarán en un archivo en formato html, el cual se podrá abrir desde el entorno de trabajo (como se ve en el menú superior de la Figura 3). Este archivo contendrá al inicio el día y hora de ejecución, seguido de una tabla que contendrá la descripción detallada de cada uno de los errores. La tabla de errores debe contener como mínimo la siguiente información:

- Línea: Número de línea donde se encuentra el error.
- Columna: Número de columna donde se encuentra el error.
- Tipo de error: Identifica el tipo de error encontrado. Este puede ser léxico, sintáctico o semántico.
- Descripción del error: Dar una explicación concisa de por qué se generó el error.

A continuación se muestra una tabla como ejemplo del archivo de salida con algunos errores:




# Errores

**Día de ejecución: 13 de Agosto de 2016**

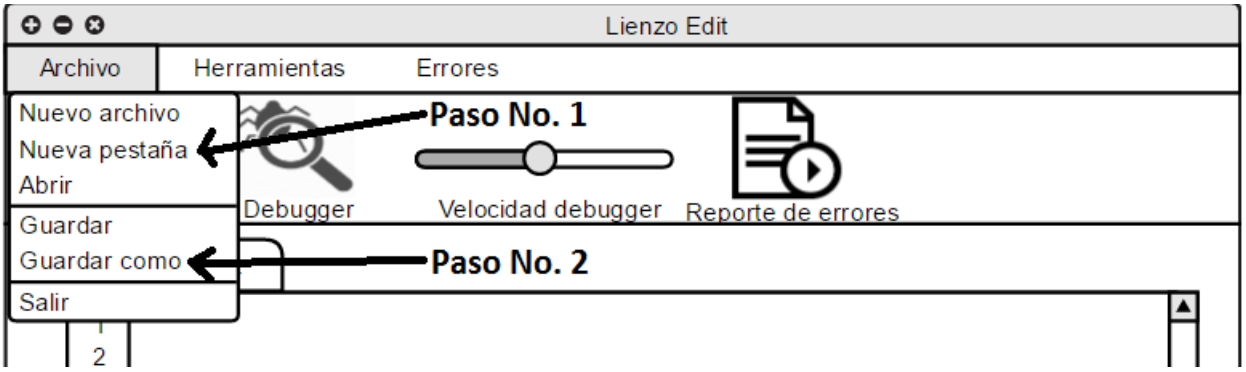
**Hora de ejecución: 12:30:15 p.m.**

| Linea | Columna | Tipo de Error | Descripcion |
|---|---|---|---|
| 14 | 8 | Lexico | Simbolo ~ no esperado |
| 68 | 5 | Sintactico | Se esperaba ? |
| 105 | 10 | Semantico | var no ha sido declarada |

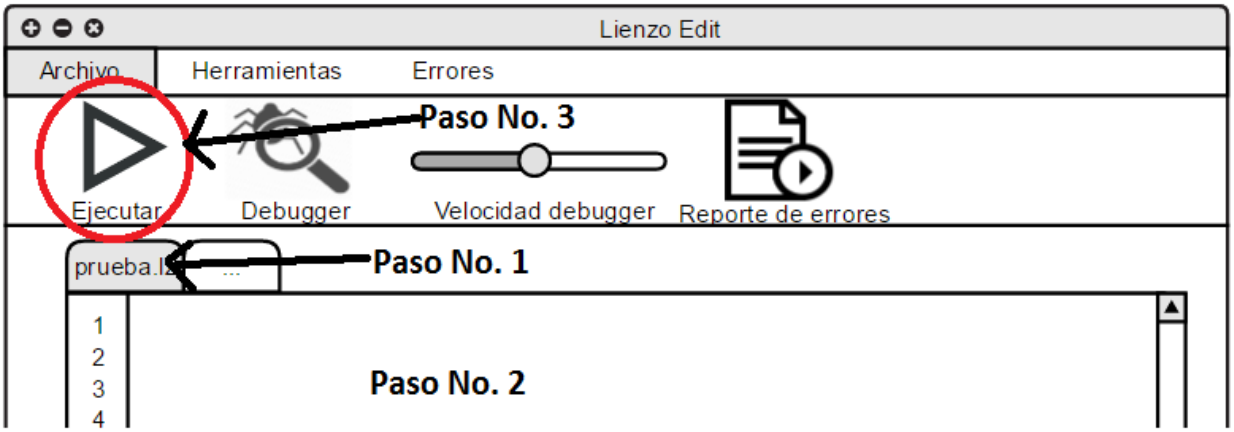
# 10. Uso de la aplicación

## Editor

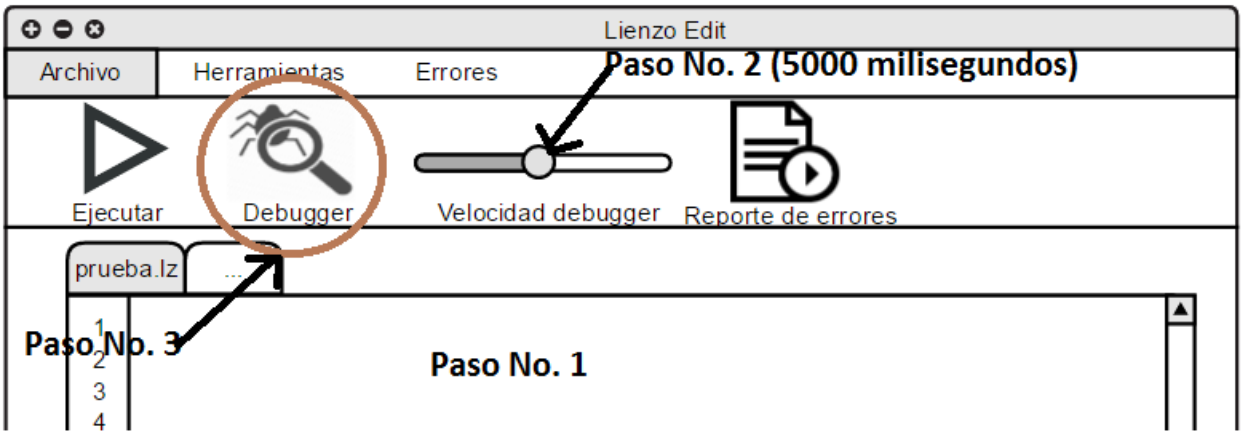
- **Crear un archivo .lz**
    1. Se crea una nueva pestaña en el editor (esta acción crea automáticamente un nuevo archivo en blanco).
    2. Se guarda el archivo como .lz (Por ejemplo lienzo1.lz).



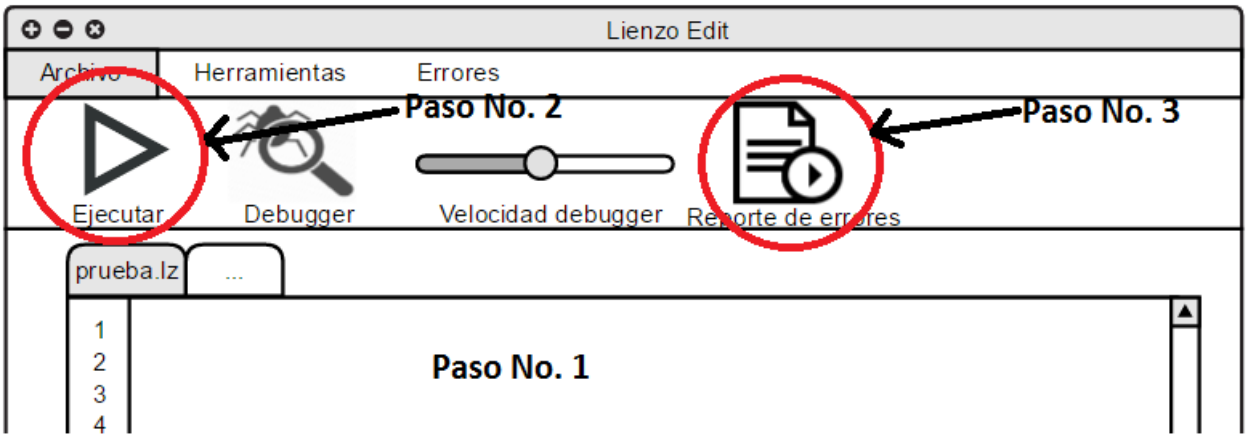
- **Ejecutar código**
    1. Se crea un archivo o archivos .lz (por ejemplo archivo “prueba.lz”).
    2. Se codifica las instrucciones en sintaxis lienzos (Lenguaje de lienzos).
    3. Se pulsa el botón ejecutar.



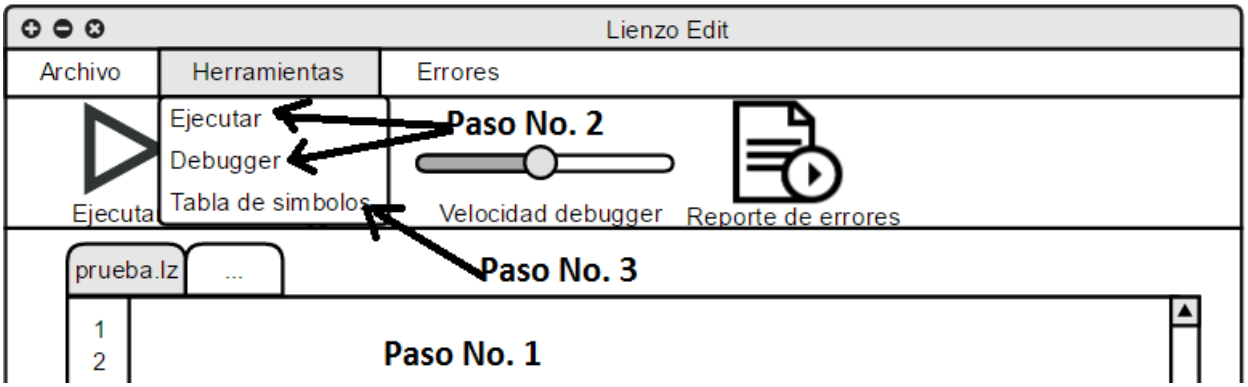

- **Depurador de código**
    1. En un archivo(s) se codifica las instrucciones en sintaxis lienzos (Lenguaje de lienzos).
    2. Mediante el selector de velocidad se indica los milisegundos que el depurador debe esperar antes de cambiar a las siguientes instrucciones.
    3. Se pulsa el botón debugger para iniciar la depuración del código (pulsar la tecla F3 para pausar/reanudar la depuración y F4 para cancelar).



- **Ver reporte de errores**
    1. En un archivo(s) se codifica las instrucciones en sintaxis lienzos (Lenguaje de lienzos).
    2. Se pulsa el botón ejecutar.
    3. Se pulsa el botón reporte de errores para ver si el intérprete detectó algún error.

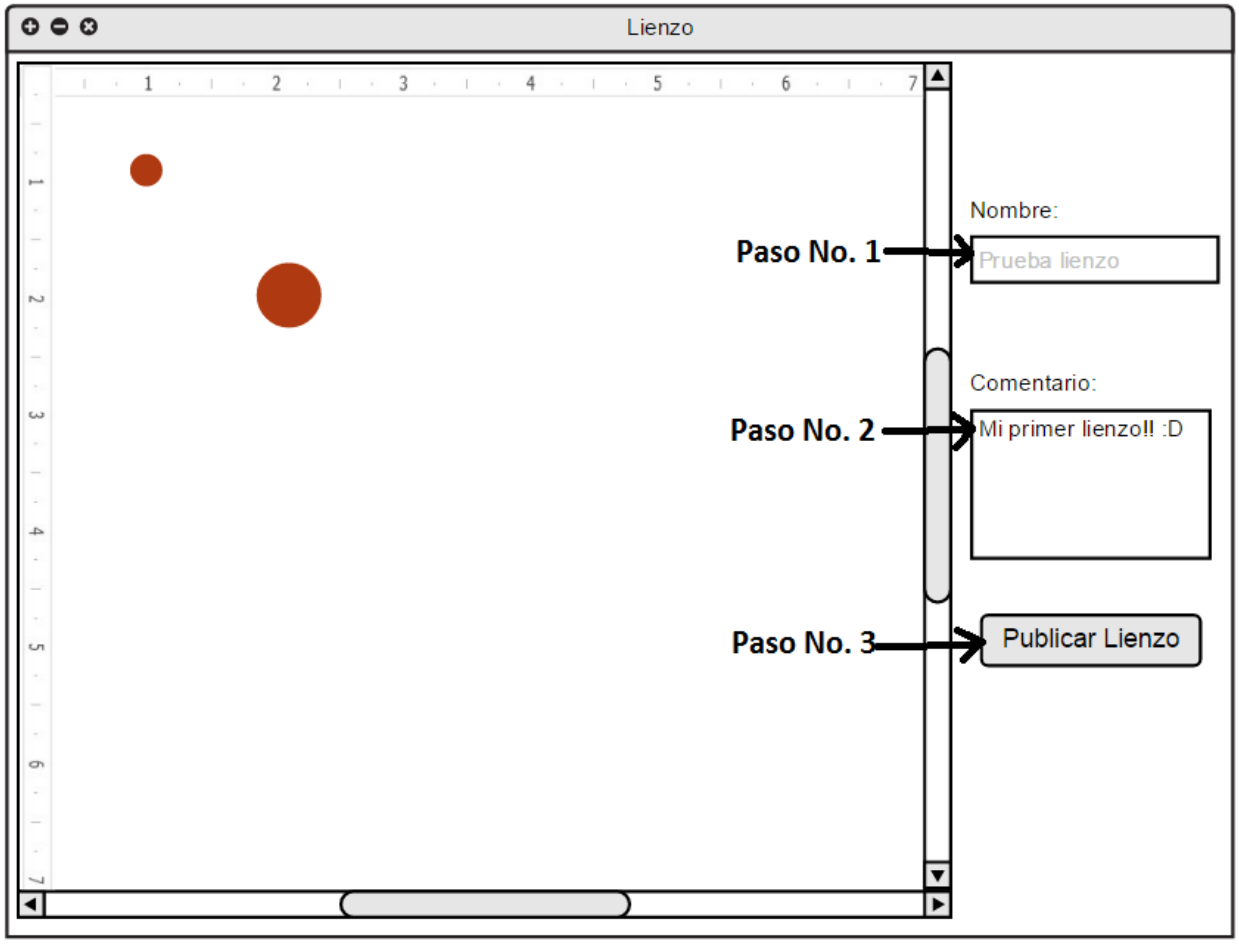



- **Ver reporte tabla de símbolos**
    1. En un archivo(s) se codifica las instrucciones en sintaxis lienzos (Lenguaje de lienzos).
    2. Se pulsa el botón ejecutar o debugger.
    3. Se pulsa el botón tabla de símbolos para generar un html con todos los símbolos.



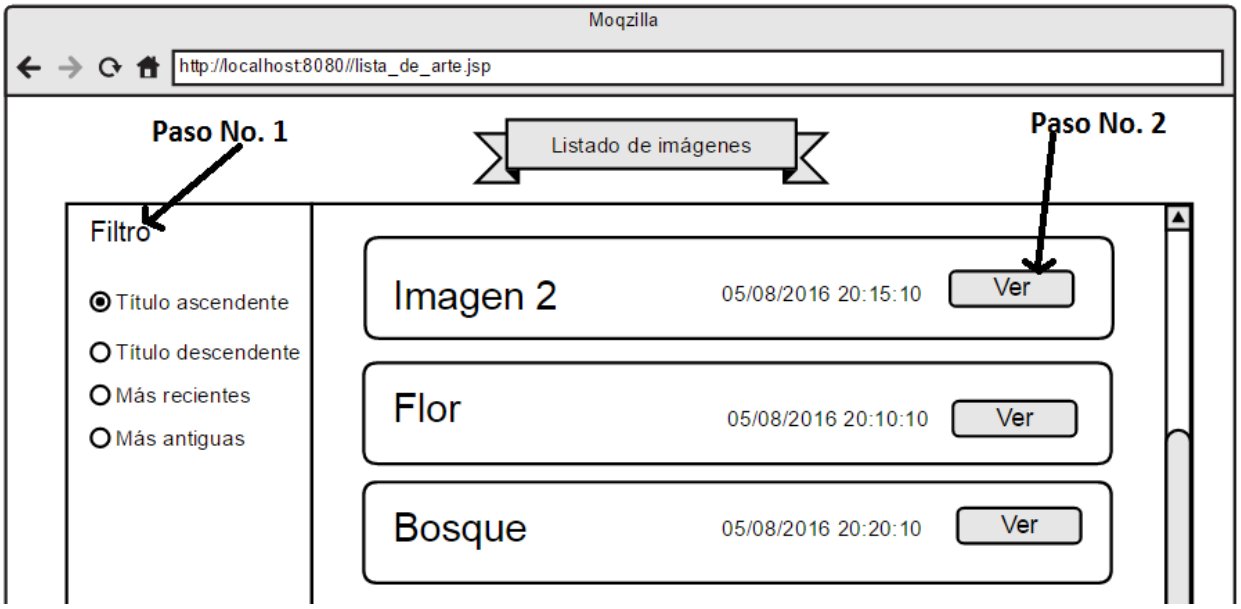
## Área gráfica del lienzo

- **Publicar lienzo**
    1. Se escribe el nombre del lienzo.
    2. Se escribe un comentario acerca del lienzo
    3. Se pulsa el botón publicar lienzo



## Galería de arte

- **Filtrar imágenes**
    1. En la sección de filtros de la galería de arte se selecciona uno de los 4 filtros existentes.
    2. Se pulsa el botón ver para visualizar una imagen.

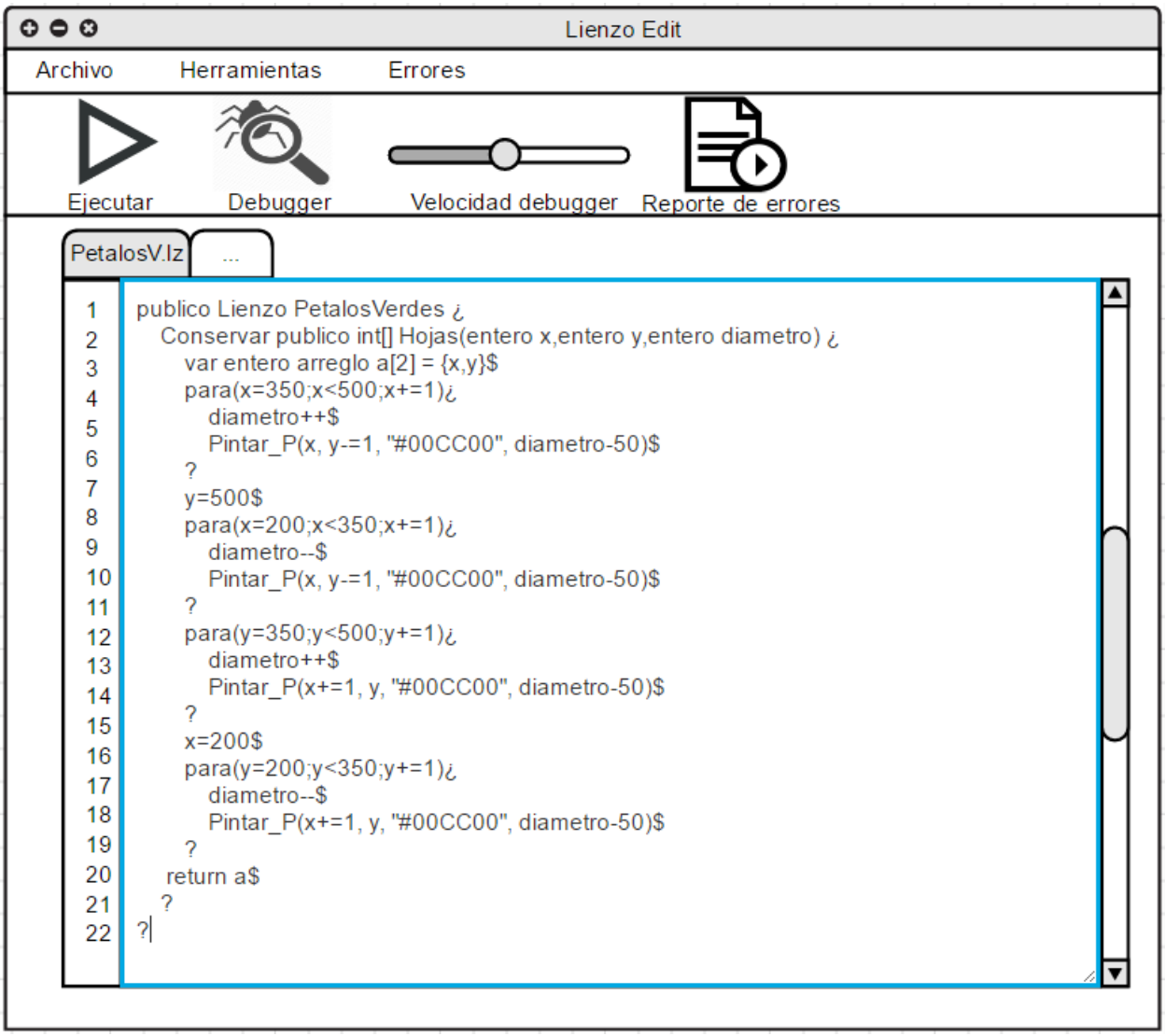



# 11. Ejemplos

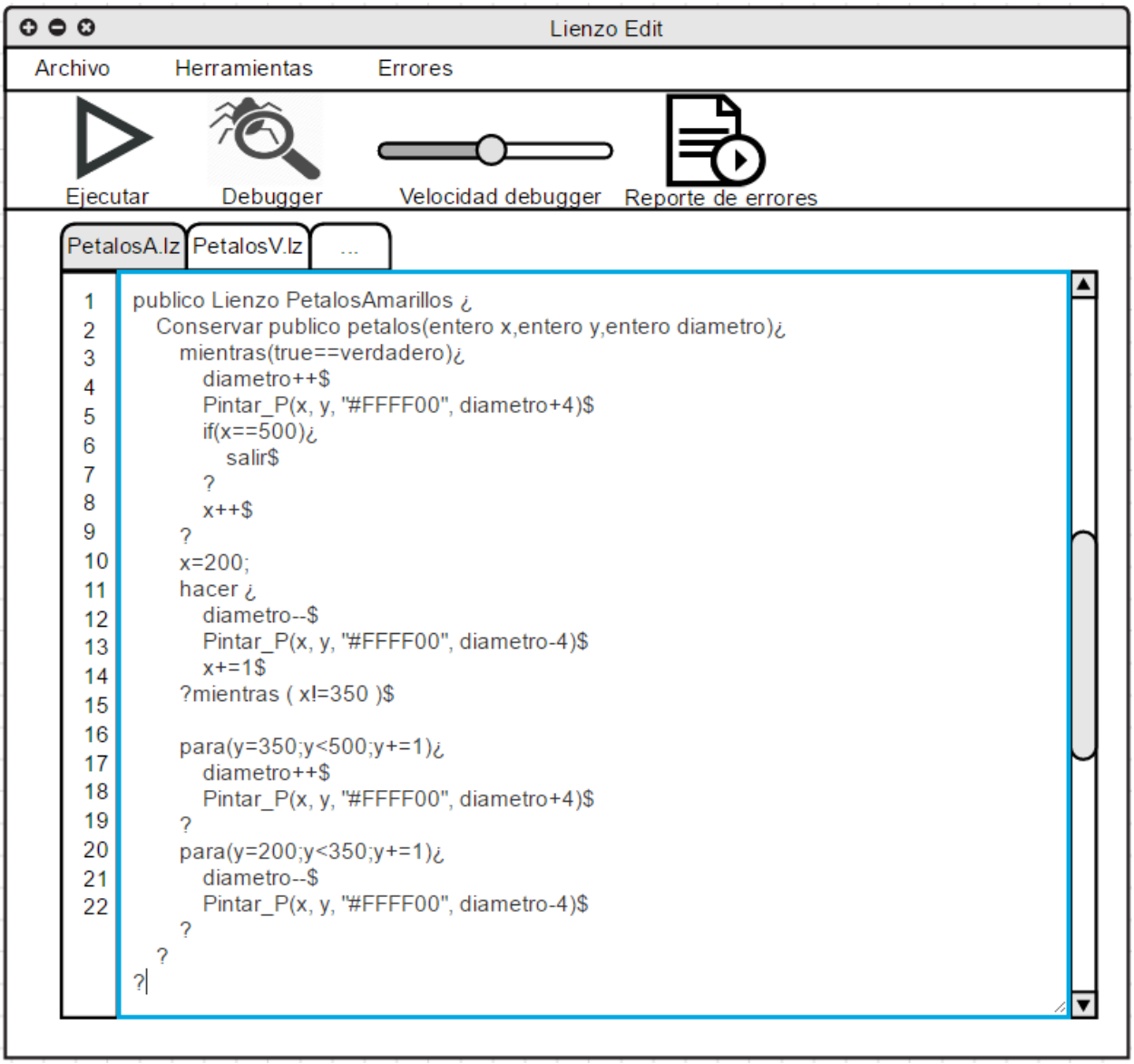
## Ejecución de código

Para este ejemplo se creará un dibujo de una Flor, se crearán 4 lienzos, el primer lienzo será para dibujar las hojas o pétalos de color verde esto se logrará definiendo el centro de la flor mediante coordenadas x & y, a partir del centro dado se dibujará una sucesión de puntos donde el último punto tendrá un diámetro ligeramente mayor el anterior dando así un efecto de pétalo (este efecto se obtiene mediante el uso de ciclos que indican la dirección y sentido de la sucesión de puntos por lo que se debe hacer 4 ciclos para crear 4 hojas de color verde). El segundo lienzo será para dibujar los pétalos de color amarillo. El procedimiento será similar al anterior con la diferencia que los pétalos amarillos tendrán dirección y sentido diferentes al anterior. El tercer lienzo será para dibujar el tronco de la flor esto se logrará con una sucesión de óvalos que van del centro de la flor hacia abajo. El cuarto lienzo será para dibujar el centro. Al final se unirán los cuatro lienzos (unión de capas) por medio de la herencia creando así un dibujo de una flor.

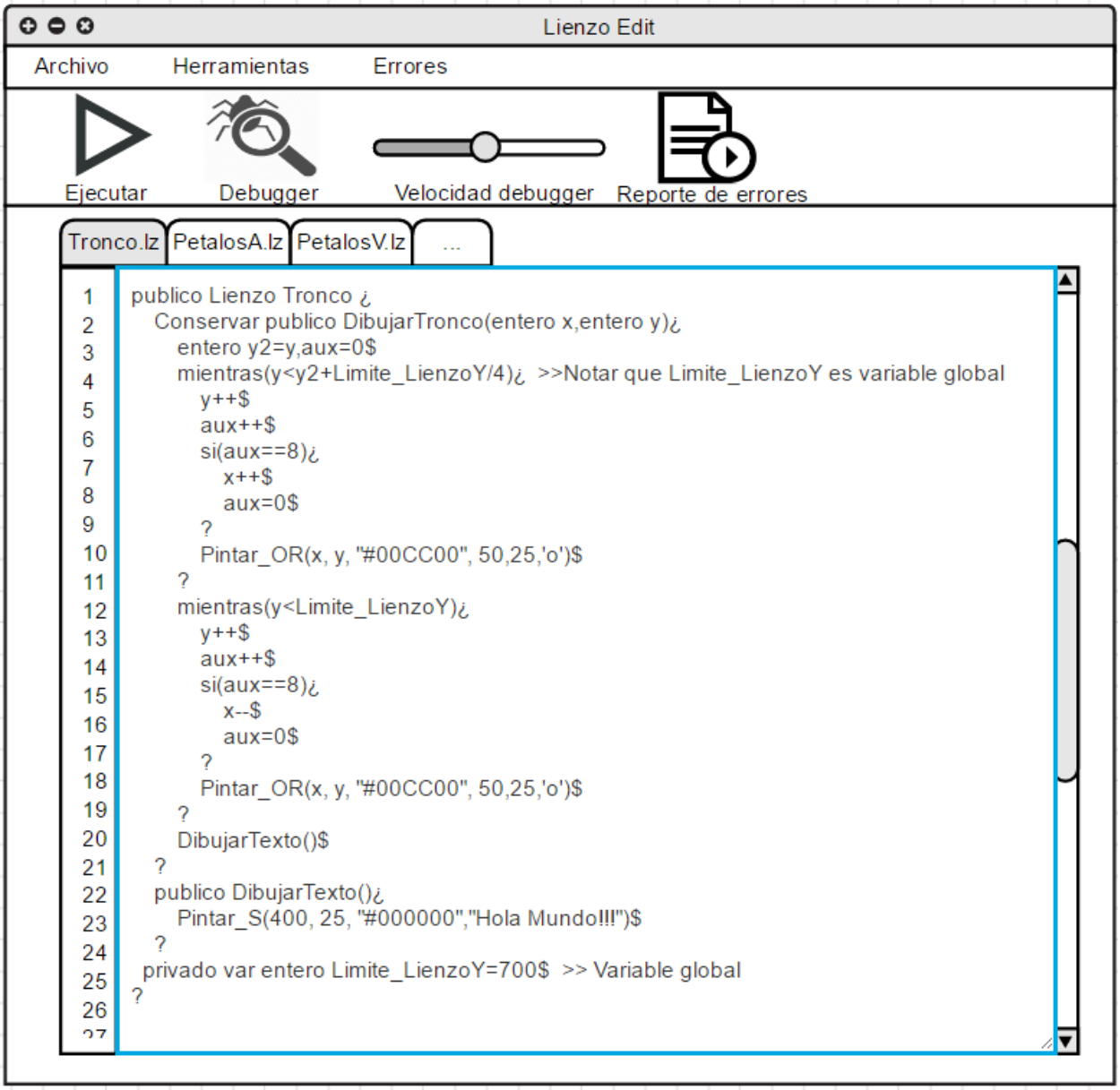
- **Creación del lienzo “PetalosVerdes”, este lienzo dibujará las hojas o pétalos verdes de la flor a una escala y posiciones X & Y proporcionadas por el usuario.**
    1. Se crea un archivo .lz con el nombre de “PetalosV.lz”.
    2. Se codifican las instrucciones tal y como se muestra a continuación.



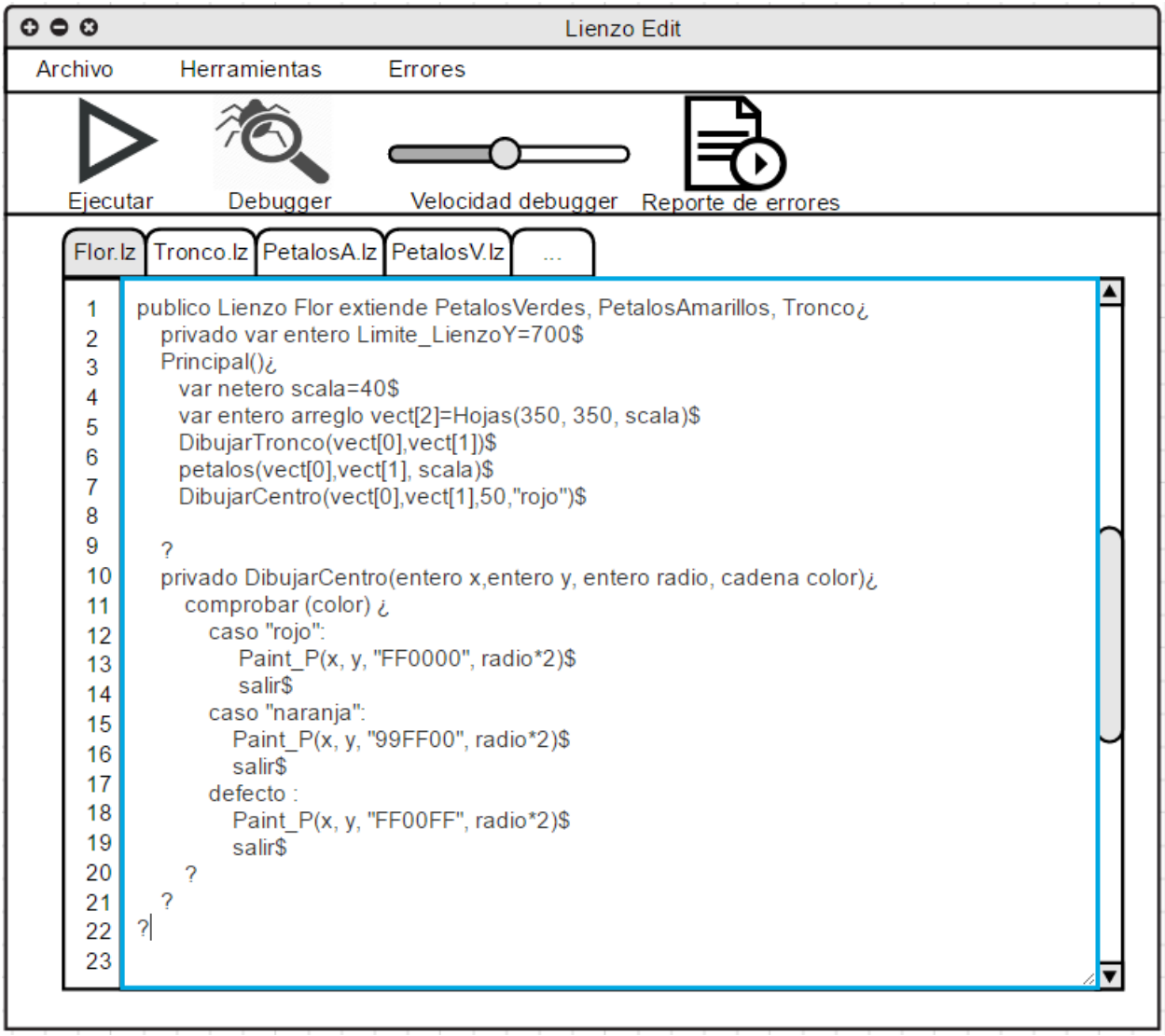
- **Creación del lienzo “PetalosAmarillos”, este lienzo dibujará los pétalos de color amarillo de la flor a una escala y posiciones X & Y proporcionadas por el usuario.**
    1. Se crea un archivo .lz con el nombre de “PetalosA.lz”.
    2. Se codifican las instrucciones tal y como se muestra a continuación.



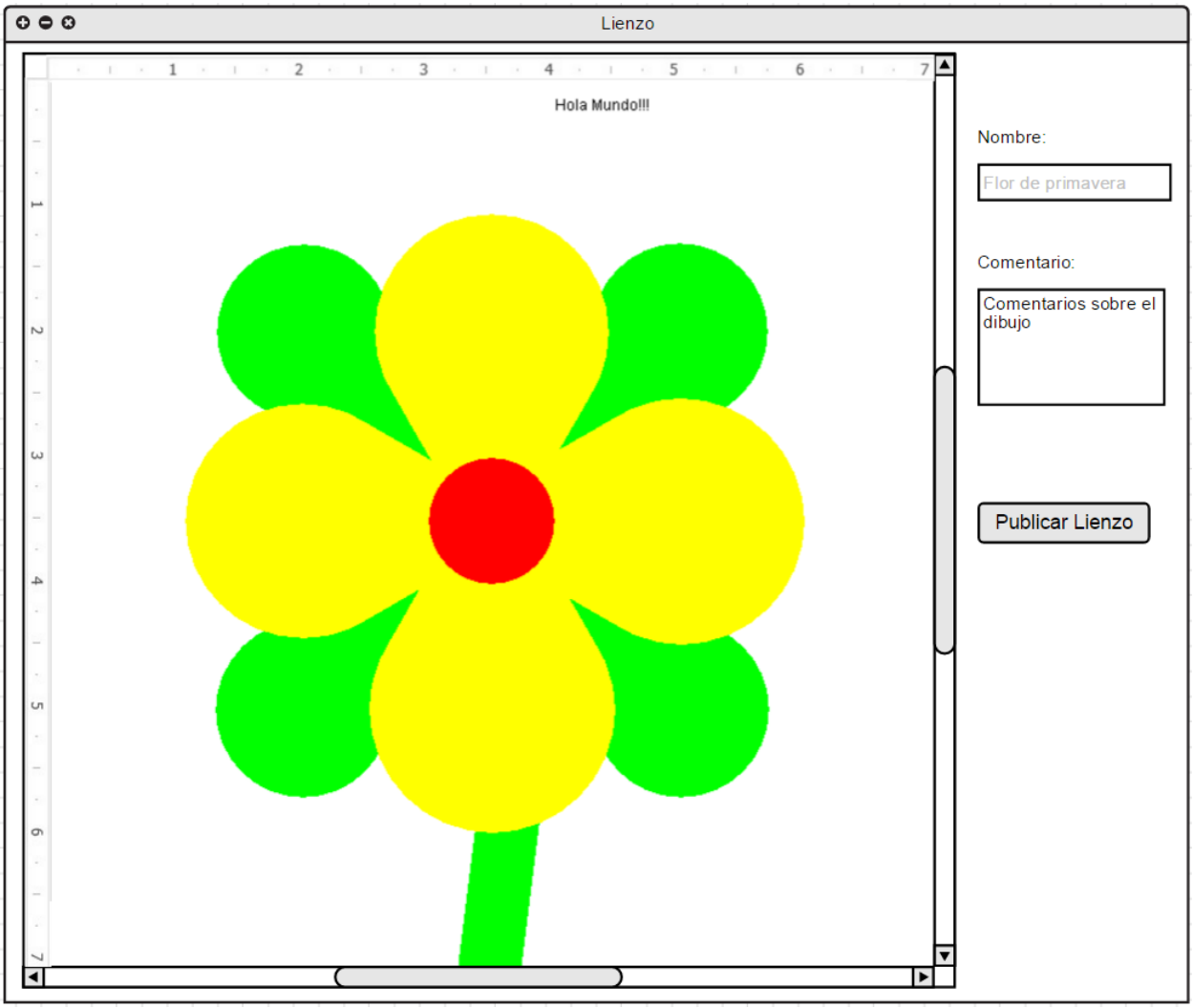
- **Creación del lienzo “Tronco”, este lienzo dibujará el tronco de la flor de color verde a una escala y posiciones X & Y proporcionadas por el usuario, además pintará un texto en una sección del área gráfica del lienzo.**
    1. Se crea un archivo .lz con el nombre de “Tronco.lz”.
    2. Se codifican las instrucciones tal y como se muestra a continuación.



- **Creación del lienzo “Flor”, este lienzo dibuja el centro de la flor de color rojo y hereda (une capas) de los lienzos PetalosAmarillos, PetalosVerdes y Tronco. Este lienzo también contiene el método principal donde inicia la ejecución del código.**
    1. Se crea un archivo .lz con el nombre de “Flor.lz”
    2. Se codifican las instrucciones tal y como se muestra a continuación.



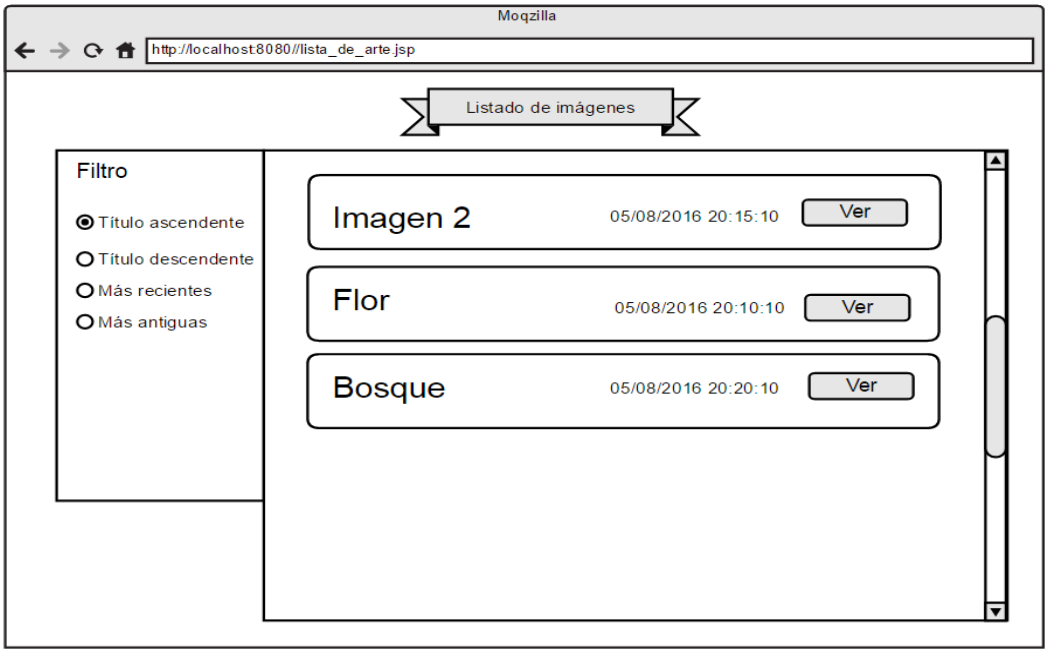
- **Resultado de la interpretación**, el intérprete es el encargado de ejecutar todo el código para generar una imagen. Notar que las posiciones X & Y ingresadas en el editor concuerda con las reglas guías del área gráfica del lienzo, obteniendo asi precisión en el dibujo.
    a. Pasos para esta acción:
        - Con el archivo que contiene el método principal (Flor.lz) abierto en el editor, se procede a ejecutar el código.
        - Se abrirá el área gráfica del lienzo mostrando lo siguiente.



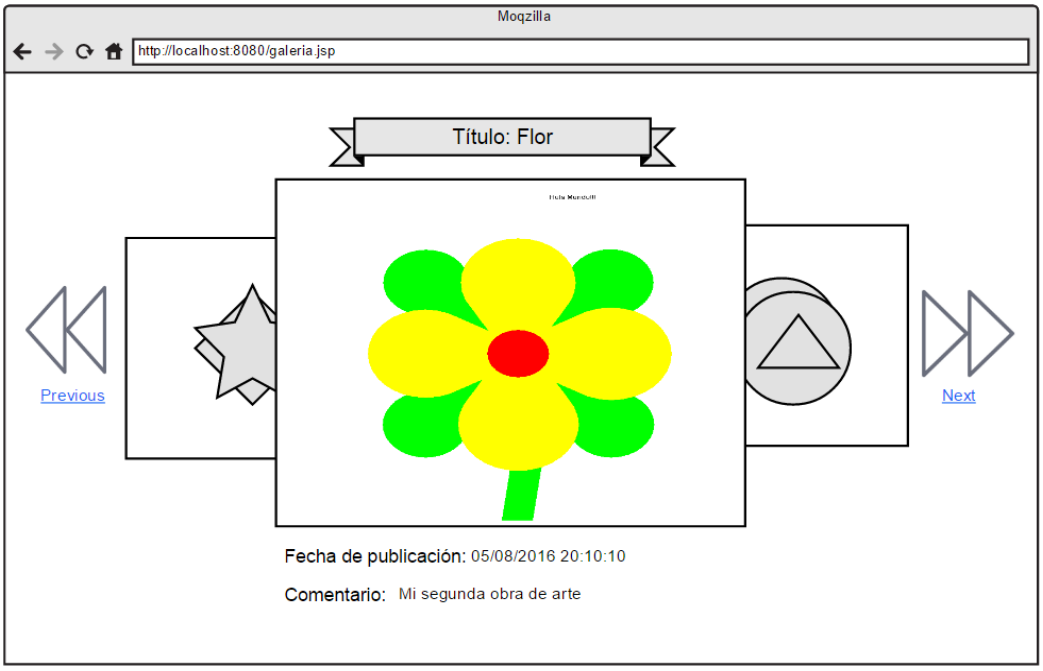
## Publicar lienzo y galería de arte

Cuando se finaliza la creación del dibujo el usuario podrá publicar su dibujo en la galería de arte.

- Pasos para esta acción:
  - En el área gráfica del lienzo ingresar el nombre del dibujo (para este ejemplo se publicara una imagen con nombre “Flor”) y un comentario del dibujo.
  - En el área gráfica del lienzo se selecciona la opción “Publicar lienzo”.
  - Se accede a la galería de arte y se muestra el siguiente menú.




- Por último se procede a observar la imagen.

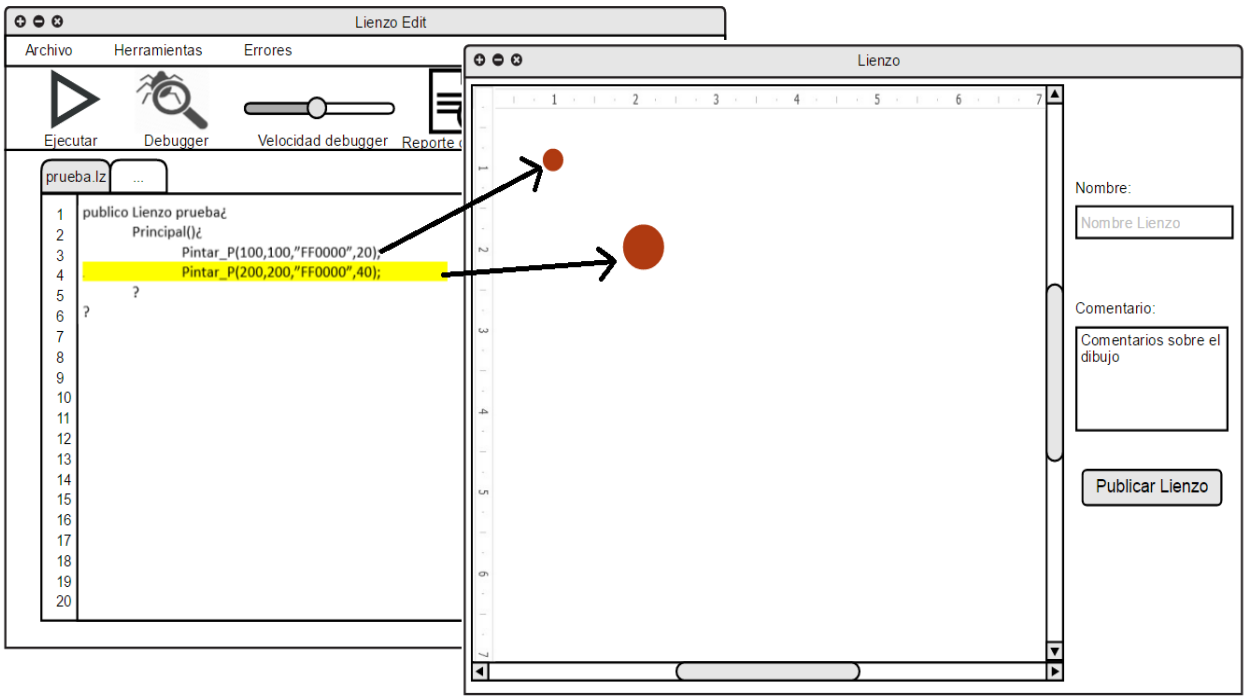


## Depurador y velocidad del depurador

Para este ejemplo se tiene el código que pinta 2 puntos con diámetros y posiciones X & Y proporcionadas por el usuario, la velocidad del debugger se encuentra configurada a 5000 milisegundos por instrucción es decir el debugger esperará 5 segundos para ejecutar la siguiente instrucción.

b. Pasos para esta acción:
   - Se crea un archivo en el editor “prueba.lz” y se ingresa el código.
   - Se establece la velocidad del debugger en este caso será de 5000 milisegundos.
   - Se ejecuta el código en modo debugger.

c. Proceso del debugger.
   - Inicia ejecutando las instrucciones del método principal.
   - Ejecuta la instrucción Pintar_P(100,100,”FF0000”,20);
   - Dibuja el punto en el área gráfica del lienzo.
   - Espera 5000 milisegundos (5 segundos).
   - Ejecuta la instrucción Pintar_P(200,200,”FF0000”,40);
   - Espera 5000 milisegundos (5 segundos).
   - Termina el debugger.



# 12. Restricciones

- La aplicación deberá desarrollarse en lenguaje Java.
- La herramienta para el analisis lexico y sintactico deberá ser JavaCC
- La galería de arte se deberá desarrollar en JSP
- La comunicación entre la aplicación y la galería de arte se limita a socket, servlet o apache thrift.
- No se permite enviar URL de las imágenes generadas a la galería de arte (se debe enviar la imagen)

# 13. Requerimientos mínimos

Los requerimientos mínimos son funcionalidades que garantizan una ejecución básica del sistema, para tener derecho de calificación se debe de cumplir con los siguientes requerimientos:

- Entorno de trabajo “Lienzo 2D”
    - Crear archivos
    - Abrir archivos
    - Ejecutar
    - Debuggear
    - Reporte de errores
    - Área gráfica del lienzo
- Código Alto nivel
    - Crear lienzos
    - Extensión de lienzos
    - Sobrecarga
    - Manejo de Procedimientos y Funciones generales
    - Funciones nativas
        - Pintar punto
        - Pintar cadena
        - Pintar óvalo y rectángulo
        - Procedimientos ordenar
        - sumarizar
    - Declaración y Asignación de variables
    - Operadores Aritméticos, Relacionales y Lógicos
    - Ciclos y Bifurcaciones
        - Si/Sino
        - Comprobar
        - Para
        - Mientras
    - Método Principal
    - Sentencia de “retorno” (todos los tipos de datos y arreglos)

- Sentencia ‘’continua”
- Sentencia salir
- Arreglos

## 14. Entregables

- Aplicación funcional.
- Código fuente.
- Gramática escrita en JavaCC.
- Manual técnico.
- Manual de usuario.

Fecha de Entrega: miércoles 21 de septiembre de 2016 antes de las 11:00 A.M.