Universidad de Santiago de chile.
Facultad de ingeniería.
Departamento de ingeniería informática.

Informe de laboratorio 2:

Simulación de una plataforma estilo "Photoshop" en lenguaje Prolog.

Nombre: Javiera Varela.
RUT: 19.957.716-6
Profesor: Roberto Gonzales.
Asignatura: Paradigmas de la programación.

Contenido

1. Introducción.

Este informe corresponde al laboratorio número 2 del curso paradigma de la programación. En este se abordará el paradigma declarativo lógico, el cual se basa en formalismos abstractos y en particular a la lógica de primer orden.

Como herramienta para poder llegar a la solución del problema, se utilizará el lenguaje de programación Prolog a través del IDE SWI-Prolog. Algo destacable dentro de este lenguaje es que puede utilizarse como la especificación de un problema, en vez de especificar pasos para llegar a la solución de un problema.

Dentro de este informe se describirá el problema, las características de este y sus limitaciones, se abordará una mayor descripción del paradigma a trabajar, se analizará el problema planteando sus requerimientos específicos que se deben cubrir, se explicara el diseño de la solución y la lógica detrás de los algoritmos creados para su realización, las instrucciones de uso, los resultados y conclusiones.

2. Descripción del problema.

El problema consta en desarrollar una simulación a una aplicación de edición de imágenes del tipo GIMP o Adobe Photoshop. Se busca la manipulación y edición de imágenes a través de un enfoque simplificado debido a las limitaciones del lenguaje a utilizar. Este proyecto se debe concentrar en la utilización de imágenes RGB o RGB-D, las cuales constan de ser una imagen con información en el espacio de colores y un espacio tridimensional.

3. Descripción del Paradigma.

El paradigma lógico pertenece a la familia de leguajes de paradigma declarativo, y está basado en a lógica simbólica, por lo tanto está orientado a las metas. El estilo de programación en este paradigma se basa en especificar que se debe lograr y no en el como

Al momento de realizar un programa en este lenguaje, se debe definir hechos, relacione y reglas, las cuales la base de conocimientos del lenguaje asume como conocidos. Este ultimo paso nos permite realizar futuras consultas dentro de la aplicación. Elementos importantes a la hora de construir un código utilizando este paradigma son:

a. Hechos: Su uso consta en definir la base de conocimientos.
b. Reglas: Es una relación implícita cuyo resultado es verdadero siempre que la premisa sea verdadera.
c. Consultas: Son aquellas preguntas que se hacen mediante la consola y donde prolog busca en la base de conocimientos para dar unas respuestas.
d. Predicados: Son aquellas cosas que queremos decir. Este se compone de variables las cuales deben ir en mayúsculas.
e. Átomo: Son aquellas cosas sobre las que se basa el conocimiento que queremos expresar.

4. Análisis del problema.

Como sé indicaba con anterioridad, el laboratorio busca crear una aplicación similar a un editor de imágenes, en el cual se pueda trabajar con este de una manera más simplificada. Para ello debemos enfocarnos en la estructura principal a trabajar, la cual consta de ser la imagen.

Antes de poder aplicar los elementos necesarios para trabajar en la aplicación, debemos definir una estructura base para poder implementar cada uno de los requerimientos básicos del programa. Para ello ser consiente de que es una imagen y en que consiste esta.

La definición técnica de una imagen digital se define como una representación de dos dimensiones en una imagen basada en una matriz numérica frecuentemente binaria. Teniendo en cuenta esto, la aplicación a crear busca añadir unos requerimientos básicos y específicos que se deben cubrir y deben trabajar con la imagen. Estos constan de ser:

- Recortar una imagen.
- Invertir una imagen horizontalmente.
- Invertir una imagen verticalmente.
- Comprimir una imagen en base a la eliminación de mayor frecuencia.
- Convertir a hexadecimal.
- Visualizar la imagen.
- Rotar la imagen 90° hacia la derecha.
- Histograma.
- Descomprimir una imagen en base a restitución del color con mayor frecuencia.
- Aplicar operaciones como las anteriores a un área seleccionada dentro de la imagen.
- Editar una imagen a partir de la aplicación de funciones especiales sobre los pixeles.
- Redimensionar imagen.

Se ha de tener en cuenta que, debido a las limitaciones del lenguaje a trabajar, esta aplicación no puede constar de una interfaz interactiva para el usuario, por lo tanto cada una de las acciones que se desea trabajar deberá tener como base el IDE SWI-Prolog e indicar que acción se desea realizar con la correspondiente consulta.

5. Diseño de la solución.

El proyecto de la simulación de un editor de imágenes en Prolog, su principal base son las listas. Para poder adecuar un diseño correcto ante cada predicado que se desea implementar, se debe tener en cuenta cada una de las reglas que el lenguaje da para trabajar con este tipo de implementación, por lo tanto se requiere del uso de recursividad.

5.1. Operación para la creación de una imagen.

Para poder implementar cada una de las operaciones, se debe crear el constructor imagen. Este constructor debe tener en consideración 3 datos principales que son otorgados por el usuario:

- Anchura. -> Entero.
- Altura. -> Entero.
- Pixeles. -> Lista.

Esto lograra que cada imagen tenga una estructura del tipo lista de la siguiente forma:

- Imagen = [Anchura, Altura, Pixeles].

Debemos tener en consideración que cada píxel debe tener una estructura propia, pues una imagen puede contener dentro de ella pixeles del tipo binarios, Hexadecimales o RGB. Para ello debemos ser conscientes de que cada píxel cuenta con una estructura propia, por lo tanto, se debe implementar una estructura que soporte cada una de sus condiciones.

Hay variables que se repiten dentro de cada estructura de un píxel, pues estas son las correspondientes a tu ubicación dentro de un plano y la profundidad correspondiente de esta. Por lo tanto cada pixel debe contener:

- X. -> Entero. Corresponde a la posición X dentro de un plano cartesiano.
- Y. -> Entero. Corresponde a la posición Y dentro de un plano cartesiano.
- Depth. -> Entero. Corresponde a la profundidad de cada pixel dentro de la imagen.

Una vez encontradas las variables que nunca cambian, debemos encontrar las variables que hacen únicos a cada pixel y lo diferencian según su composición. Para ello, un pixel del tipo pixbit debe almacenar dentro de sus valores una variable entera que almacene valores del 0 y 1, un pixel del tipo pixhex debe almacenar una estructura del tipo string ya que debe almacenar valores hexadecimales y finalmente un pixel del tipo pixrgb, el cual debe almacenar 3 valores correspondientes al rojo (R), verde (G) y azul (B), los cuales serán un entero que alcance los valores entre 0 y 255. Por lo tanto la estructura de cada pixel será:

- Pixbit = [X, Y, Bit, Depth]
- PixRGB = [X, Y, Reed, Green, Blue, Depth]
- PixHex = [X, Y, Hexadecimal, Depth]

5.2. Operaciones sobre el reconocimiento de imagen.

Para ciertas acciones y futuros predicados dentro del programa, se debe tener en consideración que tipo de imagen es la que se desea trabajar, pues algunos predicados no permiten imágenes del tipo PixBit, PixRGB o PixHex. Para ello se tiene lo siguiente:

1. **IsBitmap:**
   - Este predicado recibe como dominio una imagen y revisa las variables dentro de la estructura. Da un booleano “True” si la estructura está compuesta por Pixbits.

2. **IsRGBmap:**
   - Predicado que recibe como dominio una imagen y revisa las variables dentro de la estructura. Da un booleano "True" si la estructura esta compuesta por PixRGBs.
3. **IsHexmap:**
   - Predicado que recibe como dominio una imagen y revisa las variables dentro de la estructura. Da un booleano "True" si la estructura está compuesta por PixHexs.

5.3. Operaciones sobre la imagen.

Estas operaciones pueden ser realizadas por parte del usuario para realizar modificaciones a la estructura imagen:

1. **flipH:**
   - Permite al usuario dar vuelta la imagen de manera horizontal. La lógica tras el diseño de esta solución se basa en la posición de las coordenadas dentro de cada piel. Para ello se sigue la siguiente formula matemática:

     (Anchura – 1) – X = Nueva coordenada X.

     Una vez obtenida la nueva coordenada, esta debe ser remplazada dentro del píxel. Cabe destacar que esta solución es valida para cualquier tipo de imagen.
2. **FlipV:**
   - Sigue la misma lógica que el predicado flipH, pero en vez de trabajar con el eje X década pixel, se utiliza el eje Y. Para ello sigue la siguiente formula:

     (Altura – 1) – Y = Nueva coordenada Y.
3. **Crop:**
   - Predicado que permite cortar la imagen. Para la utilización de este predicado, el usuario debe ingresar las coordenadas (x1,y1) y (x2, y2), las cuales generan un cuadrante donde se conservan los pixeles donde sus coordenadas encajen dentro de estos límites.
4. **imageRGBtoHex:**
   - Predicado que permite transformar una imagen que contiene PixRGB a una imagen de pixeles PixHex. La lógica detrás de esta solución se basa en la comprensión de cada formato.
     Un PixRGB consta de 3 variables [Red, Green, Blue], los cuales a su vez tienen un valor dentro de las unidades decimales. Por su parte, un PixHex consta de una variable del tipo String que almacena un valor en variables hexadecimales. Por lo tanto, la transformación debe ser:

     RGB = [255,255,255] -> Hex = [FFFFFF]

     Para esto, se debe dividir el valor de cada byte en 16 del cual se obtendrá el primer dígito del byte Hex. Si la división no es exacta, el decimal sobrante se multiplicará por 16.

     255 / 16 = 15.9375
     15 = F,
     0.9375 * 16 = 15
     15 = F.
     Hex = [FF].
5. **imageToHistogram:**
   - Este es un predicado que permite la creación de un histograma encontrando el color dentro de los pixeles que más se repita.

Debemos tomar en cuenta que para la solución de este predicado, se puede obtener el valor que más se repite solamente a través de la utilización de PixBits o PixHexs. Con la creación del predicado imageRGBtoHex, este será utilizado para transformar los PixRGBs y encontrar el valor de una forma más sencilla para la máquina.

6. **Rotate90:**
   - Predicado que permite girar la imagen 90° hacia la derecha. La lógica detrás de este predicado es transformar las coordenadas de cada píxel siguiendo la rotación de una figura dentro de un plano cartesiano.

     (x,y) -> (-y,x)

     Estas nuevas coordenadas representan un giro de 90° dentro de un plano cartesiano hacia la derecha.
7. **Compress:**
   - Predicado que permite comprimir una imagen. Su planteamiento se basa en solo almacenar el color que menos se repite dentro de la imagen.
8. **changePixel:**
   - Predicado que permite cambiar un pixel dentro de una imagen. Para el optimo funcionamiento se necesita establecer las bases necesarias para su uso. Se tomo la decisión de que el predicado debe verificar el tipo de imagen y solo aceptar pixeles de remplazo de este tipo.
9. **InvertedColorRGB:**
   - Este predicado permite obtener el color simétricamente opuesto en cada canal dentro de un píxel. Un ejemplo para el color simétricamente opuesto:

     0 -> 255
     1 -> 254
     2 -> 253

     Una solución matemática para este caso es tomar el valor máximo correspondiente a 255 y restarle el valor contenido en las variables R, G y B.

     pixrgb(1, 1, 190 (R), 0 (G), 234 (B), 255, PD)
     255 - 190 = 65.
     255 - 0 = 255.
     255 - 255 = 0.

     De esta forma se planea obtener el valor opuesto del RGB.

6. Aspectos de la implementación.

6.1. Estructura del proyecto:

La estructuración del proyecto consta de una sola representación que conforma a la aplicación. Para esto se debe ser consiente que incluso la misma estructura base (imagen) es un tipo de dato en el que se utiliza para poder utilizar los demás predicados. El predicado principal de esta estructura es:

- **TDA Imagen:** Posee el constructor para la aplicación y es representado por una lista: [Anchura X Altura X Pixs]. Además presenta selectores para obtener la anchura, altura y los pixeles de manera más rapida

- **TDA PixBit:** Posee el constructor para la aplicación de un pixbit. Su representación es en base a una lista [Coordenada X, Coordenada Y, Bit, Profundidad/Depth]. Presenta dentro del código un predicado que permite identificar si la imagen ingresada es del tipo PixBit.
- **TDA PixHex:** Posee el constructor para la aplicación de un pixhex. Su representación es en base a una lista [Coordenada X, Coordenada Y, Color hexadecimal, Depth]. Presenta dentro del código un predicado que permite identificar si la imagen ingresada es del tipo PixHex.
- **TDA PixRGB:** Posee el constructor para la aplicación de un PixRGB. Su representación es en base a una lista [Coordenada X, Coordenada Y, Color rojo, Color verde, Color azul, Depth]. Presenta dentro del código un predicado que permite identificar si la imagen ingresada es una del tipo PixRGB, pero además contiene predicados que permiten cambiar sus valores de color al tipo PixHex.

Cabe destacar que el laboratorio no trabaja con archivos externos al main, tampoco se utilizaron librerías especiales, sino que solo se utilizó predicados nativos del lenguaje Prolog como length, append, maplist, entre otros.

Una vez abierto el main, podemos encontrar el código organizado de la siguiente estructura:

- Predicados no relacionados al TDA.
- TDA imagen.
- Predicados obligatorios.

El proyecta consta de un archivo aparte en el cual se almacenan los ejemplos y usos de cada predicado. Para este archivo en concreto se necesito la importación del código a través de la función:

Include('main_19957716-6_Varela_Vilches.pl').

Para este proyecto no se considero ninguna biblioteca en particular.

6.2. Compilador o interprete usado:

Para este proyecto es necesario el IDE SWI-Prolog, específicamente la versión 8.x.x o superior. Como alternativa es posible utilizar Visual Studio Code con la extensión de SWI-Prolog. Se conoce una alternativa llamada SWISH (Prolog online), pero no se a revisado si el código funciona correctamente en dicha plataforma.

7. Instrucciones de uso.

Lo primero que se debe hacer es ejecutar el código. Una vez que Prolog haya creado su base de conocimiento en base al código escrito.

Cosas para tener en consideración: Cada vez que se ingrese un string debe contener comillas simples (‘_’) y no comillas dobles (“_”) pues el programa no reconoce estas últimas como string.

Se espera errores en los siguientes predicados dentro del programa:

- Compress.
- changePixel.
- InvertedColorRGB.

Esto debido a errores a la hora de implementar la recursión y solución correctas para dichos predicados. De todas formas, el funcionamiento de otros predicados no debería verse afectados.

Como parte de los resultados esperados, se puede comprobar que funcionan con normalidad. En la parte de anexos se puede comprobar la creación de los predicados y ejemplos de estos.

- Imagen pixbit (Anexo 1).
- Imagen PixHex (Anexo 2).
- Imagen PixRGB (Anexo 3).
- Creación de un Pixbit (Anexo 4).
- Creación de un PixRGB (Anexo 5).
- Creación de un PixHex (Anexo 6).
- Uso de flipH (Anexo 7).
- Uso de flipV (Anexo 8).
- Uso de crop (Anexo 9).
- Uso de imagenRGBtoHex (Anexo 10).
- Uso de imageToHistogram (Anexo 11).
- Uso de rotate90 (Anexo 12).

8. Resultados.

Requerimientos empleados dentro del programa y su nivel de implementación:

- 100% -> El constructor o predicado se implemento de manera correcta y con los resultados solicitados.
- 50% -> El predicado se implementó y se pensó en su diseño de solución, pero el resultado dado no cumple con el objetivo solicitado.
- 0% -> El predicado directamente no fue implementado. (No se supo implementar)

Predicado.	Grado de alcance.
TDA image - Constructor	100%
TDA pixbit - Constructor	100%
TDA pixhex - Constructor	100%
TDA pixrgb – Constructor	100%
isBitmap – Predicado	100%
Ispixmap - Predicado	100%
isHex - Predicado	100%
flipH - Predicado	100%
flipV - Predicado	100%
Crop - Predicado	100%
imageRGBToHex – Predicado	100%
imageToHistogram – Predicado	100%
imageRotate90 – Predicado	100%
Compress – Predicado	50%
changePixel – Predicado	50%
invertColorRGB - Predicado	50%
Iscompressed – Predicado	0%
imageToString - Predicado	0%
imageDepthLayers - Predicado	0%
imageDecompress - Predicado	0%

Como una autoevaluación, se marcan las casillas de diferentes colores en la parte de grado de alcance, para una visualización más notoria con respecto al éxito o fracaso logrando en cada implementación de los TDA.

9. Conclusiones.

Después de realizar el proyecto, se puede concluir que se cumplió el objetivo principal de este laboratorio que consiste en aprender el lenguaje de programación Prolog y el paradigma declarativo – lógico de forma correcta, pues a pesar de que algunos predicados no se pudieron implementar como se esperaban, en la mayoría de los predicados se pudo llevar a cabo la recursividad y la solución esperada con éxito utilizando el aprendizaje que se obtuvo en clases.

La programación declarativa, especialmente la programación lógica, hace que los elementos basados en listas sean más fáciles de programar en función del que y no del cómo. Prolog se destaca por utiliza la lógica basándose en hechos o predicados declarados por el programador, es decir, si existe dentro de la base de datos del programa o no, por lo que cualquier consulta incorrecta dará como resultado falso. Las variables anónimas también logran esto al preocuparse solo por las variables que afectan dichos predicados.

Una de las principales dificultades dentro del proyecto consistía en que al ser un lenguaje diferente a lo que se acostumbra, muchas de las implementaciones de lo predicados necesitaban ser cambiadas múltiples veces antes de llegar a un resultado final coherente a lo solicitado. Este proyecto debe empezarse con anticipación, pues es un lenguaje que necesita tiempo para su comprensión.

En comparación al paradigma funcional, este paradigma resulto más corto en cuanto su realización y como estudiante se obtuvo mejores resultados. De un laboratorio donde no se pudo realizar casi ninguna función por falta de comprensión del paradigma, en este se puede ver un gran avance en cuanto a la realización del proyecto que se lleva a cabo.

10. Referencias.

1. Prolog by mppinedav. (s. f.). Recuperado 3 de noviembre de 2022, de https://ferestrepoca.github.io/paradigmas-de-programacion/proglogica/tutoriales/prolog-gh-pages/index.html
2. Collier, J. (2021, 7 mayo). RGB to Hex: Understanding the Major Web Color Codes. DevelopIntelligence. https://www.developintelligence.com/blog/2017/02/rgb-to-hex-understanding-the-major-web-color-codes/
3. SWI-Prolog. (s. f.). Recuperado 3 de noviembre de 2022, de https://www.swi-prolog.org

11. Anexos.

Anexo 1 – Imagen del tipo PixBit.

```
?-     pixbit(0, 0, 1, 10, PA),
    pixbit(0, 1, 0, 20, PB),
    pixbit(1, 0, 0, 30, PC),
    pixbit(1, 1, 1, 4, PD),
|    |    |        imagen(2, 2, [PA, PB, PC, PD], X).|
PA = [0, 0, 1, 10],
PB = [0, 1, 0, 20],
PC = [1, 0, 0, 30],
PD = [1, 1, 1, 4],
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]].
```

Anexo 2 – Imagen del tipo PixHex.

```
?-     pixhex( 0, 0, '#FF0000', 20, PA),
    pixhex( 0, 1, '#FF0000', 20, PB),
    pixhex( 0, 2, '#FF0000', 20, PC),
    pixhex( 1, 0, '#0000FF', 30, PD),
    pixhex( 1, 1, '#0000FF', 4, PE),
    pixhex( 1, 2, '#0000FF', 4, PF),
    pixhex( 2, 0, '#0000FF', 4, PG),
    pixhex( 2, 1, '#0000FF', 4, PH),
    pixhex( 2, 2, '#0000FF', 4, PI),
    imagen(3, 3, [PA, PB, PC, PD, PE, PF, PG, PH, PI], X).|    |    |    |    |
|    |    |    |    |    |    |    |        imagen(3, 3, [PA, PB, PC, PD, PE, PF
, PG, PH, PI], X).
PA = [0, 0, '#FF0000', 20],
PB = [0, 1, '#FF0000', 20],
PC = [0, 2, '#FF0000', 20],
PD = [1, 0, '#0000FF', 30],
PE = [1, 1, '#0000FF', 4],
PF = [1, 2, '#0000FF', 4],
PG = [2, 0, '#0000FF', 4],
PH = [2, 1, '#0000FF', 4],
PI = [2, 2, '#0000FF', 4],
X = [3, 3, [[0, 0, '#FF0000', 20], [0, 1, '#FF0000', 20], [0, 2, '#FF0000'|...],
 [1, 0|...], [1|...], [...|...]|...]].
```

Anexo 3 – Imagen del tipo PixRGB.

```
?-     pixrgb(0, 0, 0, 20, 123, 10, PA),
    pixrgb(0, 1, 230, 20, 13, 20, PB),
    pixrgb(1, 0, 50, 90, 143, 30, PC),
|    |        imagen(2, 2, [PA, PB, PC], X).|
PA = [0, 0, 0, 20, 123, 10],
PB = [0, 1, 230, 20, 13, 20],
PC = [1, 0, 50, 90, 143, 30],
X = [2, 2, [[0, 0, 0, 20, 123|...], [0, 1, 230, 20|...], [1, 0, 50|...]]].

?-     pixrgb(0, 0, 4, 64, 121, 10, PA),
    pixrgb(0, 1, 20, 23, 64, 20, PB),
    pixrgb(1, 0, 70, 34, 32, 30, PC),
    pixrgb(1, 1, 10, 190, 83, 30, PD),
|    |    |        imagen(2, 2, [PA, PB, PC, PD], X).|
PA = [0, 0, 4, 64, 121, 10],
PB = [0, 1, 20, 23, 64, 20],
PC = [1, 0, 70, 34, 32, 30],
PD = [1, 1, 10, 190, 83, 30],
X = [2, 2, [[0, 0, 4, 64, 121|...], [0, 1, 20, 23|...], [1, 0, 70|...], [1, 1|...]]].
```

Anexo 4 – Creación de un Pixbit.

```
?-      pixbit(0, 0, 1, 10, PA),
|         isPixbit(PA).
PA = [0, 0, 1, 10].

?-      pixbit( 0, 0, '#FF0000', 20, PA),
|         isPixbit(PA).
false.

?-      pixbit(0, 0, 200, 200, 200, 10, PA),
|         isPixbit(PA).
ERROR: Unknown procedure: pixbit/7
ERROR:      However, there are definitions for:
ERROR:          pixbit/5
false.
```

Anexo 5 – Creación de un PixRGB

```
?-      pixrgb(0, 0, 200, 200, 200, 10, PA),
|         isPixRGB(PA).
PA = [0, 0, 200, 200, 200, 10].

?-      pixrgb( 0, 0, '#FF0000', 20, PA),
|         isPixRGB(PA).
ERROR: Unknown procedure: pixrgb/5
ERROR:      However, there are definitions for:
ERROR:          pixrgb/7
false.

?-      pixrgb(0, 0, 1, 10, PA),
    isPixRGB(PA).
|     ERROR: Unknown procedure: pixrgb/5
ERROR:      However, there are definitions for:
ERROR:          pixrgb/7
false.
```

Anexo 6 – Creación de un PixHex.

```
?-
|         pixhex( 0, 0, '#FF0000', 20, PA),
|         isHex(PA).
PA = [0, 0, '#FF0000', 20].

?-      pixhex(0, 0, 1, 10, PA),
|         isHex(PA).
false.

?-      pixhex(0, 0, 200, 200, 200, 10, PA),
|         isHex(PA).
ERROR: Unknown procedure: pixhex/7
ERROR:      However, there are definitions for:
ERROR:          pixhex/5
false.
```

Anexo 7 – Uso de flipH.

```
?-      ejemploImage(X),
|         flipH(X, Y).
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]],
Y = [2, 2, [[1, 0, 1, 10], [1, 1, 0, 20], [0, 0, 0|...], [0, 1|...]]].
```

Anexo 8 – Uso de flipV

```
?-      ejemploImage(X),
|         flipV(X, Y).
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]],
Y = [2, 2, [[0, 1, 1, 10], [0, 0, 0, 20], [1, 1, 0|...], [1, 0|...]]].
```

Anexo 9 – Uso de crop.

```
?-
|         ejemploImage(X),
|         crop(X, 0, 0, 1, 1, Y).
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]],
Y = [1, 1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]].
```

Anexo 10 – Uso de imagenRGBtoHex

```
?-      ejemploImage3(X),
|         imageRGBtoHex(X, Y).
X = [2, 2, [[0, 0, 200, 200, 200|...], [0, 1, 200, 200|...], [1, 0, 190|...], [1, 1|...]]],

Y = [2, 2, [[0, 0, 'C8C8C8', 10], [0, 1, 'C8C8C8', 20], [1, 0, 'BEBEBE'|...], [1, 1|...]]].
```

Anexo 11 – Uso de imageToHistogram

```
?-      ejemploImage(X),
|         imageToHistogram(X, Y).
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]],
Y = [[0, 2], [1, 2]].
```

Anexo 12 – Uso de rotate90

```
?-      ejemploImage(X),
|         rotate90(X, Y).
X = [2, 2, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0|...], [1, 1|...]]],
Y = [2, 2, [[0, 0, 1, 10], [-1, 0, 0, 20], [0, 1, 0|...], [-1, 1|...]]].
```