AMD
Grado en Ingeniería Informática
3º Curso

Práctica 2: AFD y AFND.

OBJETIVOS

El objetivo principal de esta práctica es desarrollar un simulador de Autómatas Finitos Determinista (AFD) y de Autómatas Finitos No Determinista (AFND) en Java, **de forma que las transiciones del Autómata que pretendamos simular puedan ser dadas por teclado o leídas desde un fichero.**

Para ello se implementará una aplicación gráfica que debe permitir la lectura de un fichero con la especificación de un Autómata Finito. La aplicación incluirá un campo para introducir la cadena de entrada a analizar.

Una vez seleccionados el autómata y la cadena de entrada, la aplicación debe simular el comportamiento del autómata, realizando las transiciones correspondientes y mostrando el estado del autómata en cada momento y el resultado final de la simulación (ya sea la aceptación o el rechazo de la cadena).

Autómatas Finitos

Un Autómata Finito es una máquina de estados formada por cinco componentes:

- Un conjunto finito de estados.
- Un conjunto finito de símbolos que pueden ser utilizados como alfabeto de entrada. La entrada del autómata estará formada por una secuencia finita de símbolos del alfabeto.
- Un conjunto finito de transiciones, que están formadas por el estado origen, el estado destino y el símbolo del alfabeto que activa la transición.
- Un estado inicial, perteneciente al conjunto de estados, que describe el estado del autómata antes de comenzar a analizar la cadena de entrada.
- Un conjunto de estados finales que corresponde a un subconjunto de los estados del autómata. La cadena de entrada se aceptará si el autómata queda en un estado final después de las transiciones correspondientes a los símbolos de la cadena de entrada.

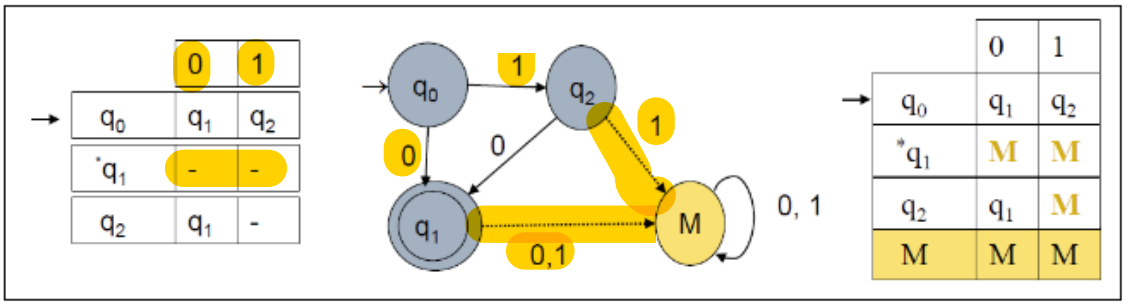
El funcionamiento de un Autómata Finito es el siguiente:

- Al comienzo del análisis el autómata se encuentra en el estado inicial.
- El autómata lee de forma secuencial los símbolos de la cadena de entrada. Cada símbolo provoca un cambio de estado del autómata, correspondiente a la transición indicada por el estado actual y el símbolo leído.
- Al terminar el recorrido de la cadena de entrada, si el estado del autómata es un estado final la cadena se ACEPTA. Si el estado no es un estado final, la cadena se RECHAZA.

Los Autómatas Finitos se pueden representar:

- **en forma de tabla:** asignando una fila a cada estado (marcando el estado inicial y los estados finales) y una columna a cada símbolo. Las celdas de la tabla indican el estado destino de las transiciones.

- **en forma diagramática:** dada estado se representa como un círculo y las transiciones se representan como arcos etiquetados. El estado inicial se indica con una flecha y los estados finales se denotan con un círculo doble.



Es frecuente que en la especificación de un autómata no aparezcan todas las transiciones, apareciendo celdas vacías en la representación tabular. Se considera que todas las transiciones no definidas son transiciones a un estado muerto.

M= Estado muerto

Autómatas Finitos Deterministas (AFD) y No Deterministas (AFND)

- Los autómatas descritos anteriormente se denominan Autómatas Finitos Deterministas (AFD) porque dado un estado y un símbolo sólo puede existir una transición (o ninguna si no consideramos el estado muerto). Esto se verifica en la representación tabular viendo que en las celdas solo puede haber un estado (o ninguno si no consideramos el estado muerto).

- Se denominan Autómatas Finitos No Deterministas (AFND) a los autómatas que no verifican la condición anterior, es decir, que pueden tener transiciones con el mismo origen y símbolo que dirijan a diferentes estados de destino. En forma tabular esto quiere decir que en cada celda habría un conjunto de estados de destino. Los AFND pueden tener además transiciones lambda, que son transiciones que no consumen ningún símbolo. Esto quiere decir que el autómata puede cambiar de estado sin leer ningún carácter en la cadena de entrada.

AFND

El funcionamiento de un AFD se puede observar en la siguiente figura:

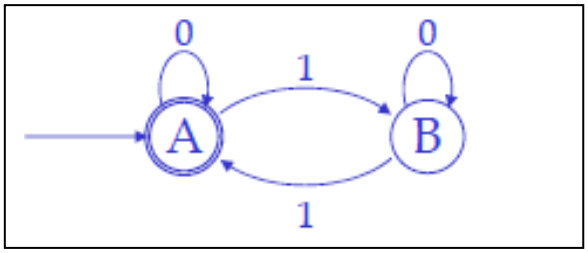


Figura 1: Ejemplo de AFD

En un AFD sólo existe un camino posible, es decir, dada una cadena sólo es posible llegar a un estado a partir del estado inicial, ya que en los AFD dado un estado y un símbolo sólo puede existir una transición (o ninguna si consideramos el estado muerto)

A modo de ejemplo, una posible implementación simplificada de un AFD puede ser la siguiente (el alumno puede utilizar las estructuras de datos y clases Java que considere más cómodas y oportunas).

AFD

```
package libClases;

public class AFD {
  private int [ ] estadosFinales; //indica cuales son los estados Finales
  private List<TransicionAFD> transiciones; //indica la lista de transiciones del AFD

  public AFD();
  public agregarTransicion(int e1, char simbolo, int e2);
  public int transicion(int estado, char simbolo);
  public boolean esFinal(int estado):

  public boolean reconocer(String cadena)  {
        char[ ] simbolo = cadena.toCharArray();
        int estado = 0 ;  //El estado inicial es el 0
        for(int i=0; i<simbolo.length; i++) {
              estado = transicion(estado,simbolo[i]);
        }
        return esFinal(estado);
  }

  public static AFD pedir();
}
```

La clase AFD describe un autómata finito determinista genérico mediante un estado inicial, una lista de transiciones y un conjunto de estados finales. Los estados se representan mediante números enteros, los símbolos mediante caracteres y las transiciones mediante tuplas de 3 elementos: (estado_origen, símbolo, estado_destino). Para representar el estado inicial se considera siempre el valor 0, por lo que no será necesario un atributo para indicar cuál es el estado inicial.

- **agregarTransicion(int e1, char simbolo, int e2):** añade la transición indicada a la lista de transiciones.
- ***int transicion(int estado, char simbolo):*** dado un estado y un símbolo devuelve el estado destino de la transición o -1 en caso que no exista dicha transición.
- ***boolean esFinal(int estado):*** devuelve true si estado es un estado final,.
- ***boolean reconocer(String cadena):*** true si la cadena pertenece al lenguaje descrito en el AFD.
- ***static AFD pedir():*** Pide por teclado (o lee de un ficheros) las transiciones y estados finales del AFD que se desea implementar.

AFND

Mientras que en un AFD sólo existe un camino posible, es decir, dada una cadena sólo es posible llegar a un estado a partir del estado inicial (dado un estado y un símbolo sólo puede existir una transición), en un AFND pueden existir varios caminos, de forma que una cadena es aceptada si al menos uno de los caminos conduce a un estado final (puede haber transiciones con el mismo origen y símbolo que dirijan a diferentes estados de destino).

Además, los AFND pueden tener transiciones lambda, que son transiciones que no consumen ningún símbolo. Esto quiere decir que el autómata puede cambiar de estado sin leer ningún carácter en la cadena de entrada.

Funcionamiento de un Autómata Finito No Determinista:

- El funcionamiento de un AFND se basa en que el estado del autómata en un instante se representa como un subconjunto de estados y no como un estado individual.

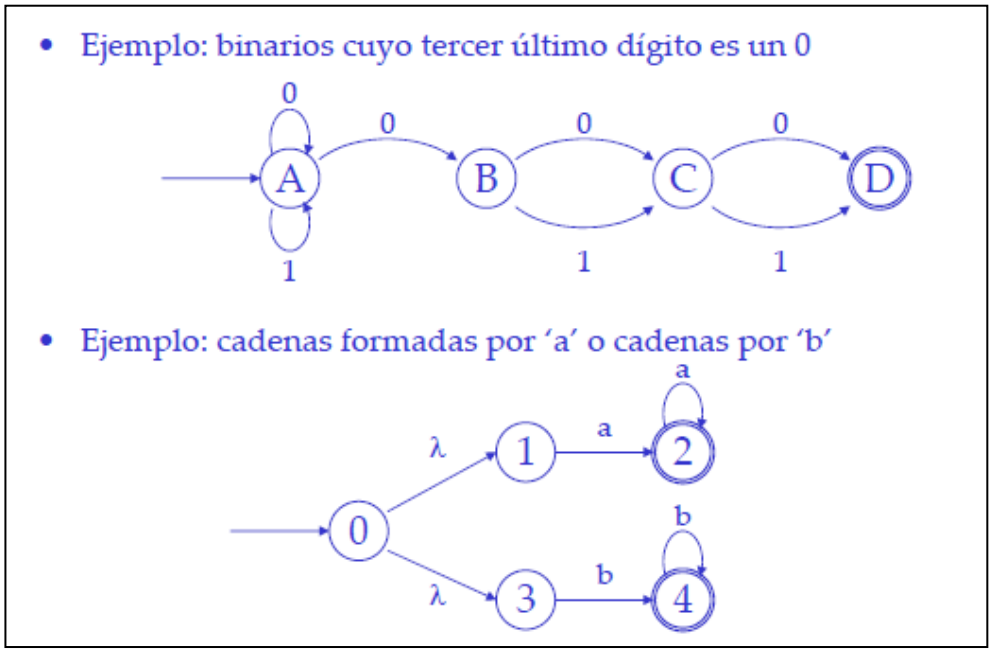
- Se denomina clausura-lambda de un conjunto de estados al resultado de añadirle a dicho conjunto todos aquellos estados que puedan alcanzarse mediante transiciones lambda.

- Al comienzo del análisis, el Autómata se encuentra en el subconjunto de estados formado por la clausura lambda del estado inicial.

- Las transiciones del Autómata calculan todos los estados accesibles desde los estados del subconjunto origen. El resultado final es la clausura lambda del conjunto de estados construido por las transiciones individuales.

- La cadena de entrada se acepta cuando el subconjunto de estados al terminar el análisis contiene algún estado final.

El funcionamiento de un AFND se puede observar en las siguientes figuras:



Como podemos observar, en un AFND pueden existir varios caminos, de forma que una cadena es aceptada si al menos uno de los caminos conduce a un estado final.

Dado un conjunto de estados E, se denomina λ-clausura(E) al conjunto de estados formado por todos los estados de E más aquellos estados a los que se puede acceder desde E mediante transiciones λ.

El macroestado inicial del AFN es λ-clausura (q0).

Partiendo de un macroestado E_i, el resultado de una transición con el símbolo 'a' es un macroestado E_{i+1} = λ-clausura (E'), donde E' está formado por todos aquellos estados que se puedan alcanzar con transiciones con el símbolo 'a' a partir de alguno de los estados de E_i

A modo de ejemplo, una posible implementación simplificada de un AFND puede ser la siguiente (el alumno puede utilizar las estructuras de datos y clases Java que considere más cómodas y oportunas).

AFND

```
package libClases;

public class AFND {
  private int [ ] estadosFinales; //indica cuales son los estados Finales
  private List<TransicionAFND> transiciones; //indica la lista de transiciones del AFND
  private List<Transicionλ> transicionesλ; //indica la lista de transiciones λ del AFND

  public AFND();
  public agregarTransicion(int e1, char simbolo, int [ ] e2);
  public agregarTransicionλ(int e1, int [ ] e2);
  private int [ ] transicion(int estado, char simbolo);
  public int [ ] transicion(int [ ] macroestado, char simbolo);
  public int [ ] transicionλ (int estado) ;
  private boolean esFinal(int estado);
  public boolean esFinal(int [ ] macroestado);
  private int[ ] λ_clausura(int[ ] macroestado);

  public boolean reconocer(String cadena)  {
          char[ ] simbolo = cadena.toCharArray();
          int [ ] estado = { 0 };  //El estado inicial es el 0
           int[ ] macroestado = λ_clausura(estado);
          for(int i=0; i<simbolo.length; i++) {
                  macroestado = transicion(macroestado, simbolo[i]);
          }
          return esFinal(macroestado);
  }

  public static AFND pedir():
}
```

La clase AFND describe un autómata finito no determinista genérico mediante un estado inicial, una lista de transiciones y transiciones-λ y un conjunto de estados finales. Los estados se representan mediante números enteros, los símbolos mediante caracteres, las transiciones mediante tuplas de 3 elementos: (estado_origen, símbolo, estados_destinos[ ]) y las λ transiciones mediante tuplas de 2 elementos: (estado_origen, estados_destinos[ ]).

En un AFND las transiciones pueden consumir un símbolo o no (transiciones lambda). El método transicion() describe las transiciones del autómata y genera como salida la lista de estados a los que se puede avanzar consumiendo un símbolo. El método transicionλ() describe las transiciones λ de cada estado, es decir, la lista de estados a los que se puede llegar por medio de transiciones que no consumen ningún símbolo.

Los macroestados del autómata se representan como listas de estados (int[ ]). El método λ_clausura() se utiliza para añadir a un conjunto de estado todos aquellos estados que sean accesibles por medio de transiciones λ.

CONSIDERACIONES A TENER EN CUENTA

Tanto la clase AFD como la clase AFND deben implementar la interfaz Cloneable (véase anexo para aquellos alumnos que no sepan qué significa) y la interfaz Proceso definida a continuación.

```
package libClases;

public interface Proceso {
  public abstract boolean esFinal(int estado); //true si estado es un estado final
  public abstract boolean reconocer(String cadena) ; //true si la cadena es reconocida
  public abstract String toString( ) ; //muestra las transiciones y estados finales
}
```

FICHEROS DE ENTRADA

Para representar los Autómatas Finitos vamos a utilizar ficheros de texto con el formato indicado a continuación:

- El conjunto de estados se describe en una línea que comienza con la cadena "ESTADOS:" seguida de la lista de identificadores de estado separados por espacios.

- El estado inicial se describe en una línea que comienza con la cadena "INICIAL:" seguida del identificador del estado inicial.

- El conjunto de estados finales se describe en una línea que comienza con la cadena "FINALES:" seguida de la lista de identificadores de estados finales separados por espacios.

- El conjunto de transiciones se describe en un bloque de texto que comienza con la línea "TRANSICIONES:" y termina con la línea "FIN".

- Cada transición se describe en una línea de texto que comienza con el identificador del estado origen, seguido del símbolo que provoca la transición y del identificador del estado de destino, separados por espacios.

- La descripción de los símbolos es la utilizada comúnmente para representar un literal de tipo carácter. Los símbolos normales se describen entre comillas simples (por ejemplo, 'a'). Para denotar caracteres no imprimibles se utiliza la barra invertida (por ejemplo, el salto de línea se describe como '\n').

A continuación, se muestra un ejemplo de especificación de un AFD.

```
ESTADOS: q0 q1 q2
INICIAL: q0
FINALES: q2
TRANSICIONES:
 q0  '0'  q1
 q0  '1'  q2
 q2  '0'  q1
FIN
```

DESARROLLO DE LA PRÁCTICA

Como se ha dicho al principio del enunciado, el objetivo principal de esta práctica es desarrollar un proyecto en lenguaje JAVA que simule el funcionamiento de un Autómata Finito Determinista (AFD) y un Autómata Finito No Determinista (AFND), de forma que las transiciones del Autómata que pretendamos simular puedan ser dadas por teclado o leídas de un fichero (descrito en el formato comentado anteriormente).

El programa desarrollado deberá tener una funcionalidad que permita leer el autómata implementado de cualquier fichero que le indiquemos (el programa debe proporcionar una opción de menú o una ventana de diálogo que permita al usuario indicar la ruta del fichero que desea abrir).

El programa deberá incluir un campo de texto para poder introducir la cadena de entrada deseada.

El programa deberá mostrar la información respecto al estado del autómata en cada instante y el resultado final (ACEPTAR - RECHAZAR) de la simulación.

El programa deberá mostrar un botón que permita simular el comportamiento del Autómata paso a paso y otro botón que permita simular el comportamiento del Autómata de una sola vez.

De forma opcional, se valorará también que la aplicación muestre por pantalla de forma gráfica las transiciones que el autómata ha realizado para reconocer la cadena introducida (en caso que dicha cadena sea reconocida por el autómata).

DOCUMENTACIÓN A ENTREGAR

La documentación a entregar para esta práctica debe incluir la siguiente información:

1. Código fuente de todas las clases desarrolladas, debidamente comentado.
2. Documentación de todas las clases usadas mediante la herramienta Java-Doc.
3. Memoria explicativa de cómo se ha implementado la práctica, cuáles son los principales problemas encontrados y modo en el que lo han solucionado.
4. Descripción de un conjunto de autómatas de prueba, que cubran todas las posibles situaciones planteadas, y que deberán de adjuntar a la memoria.

LA PRÁCTICA PODRÁ SER REALIZADA EN GRUPO CON UN MÁXIMO DE 2 COMPONENTES (en ese caso ambos alumnos deben subir la documentación y en la portada se debe indicar los nombres de los integrantes del grupo).

La defensa de la práctica y la comprobación de su correcto funcionamiento se realizará el mismo día de entrega de la práctica.

ANEXO: DUPLICACIÓN DE OBJETOS

A veces un objeto que invoca un método o que se pasa como parámetro a un método se modifica en el curso de la llamada. Si interesa que el objeto no sufra cambios podemos realizar una copia del objeto original y realizar las trasformaciones en la copia.

EL INTERFACE Cloneable

Si se desea poder clonar una clase hay que implementar la interface ***Cloneable*** y redefinir el método ***clone()*** de Object. El método original de ***Object devuelve un objeto idéntico*** haciendo una copia binaria de los atributos (los atributos de tipos primitivos copian sus valores y los atributos referencias también, haciendo que las referencias original y copia apunten al mismo objeto) y lanza una excepción *CloneNotSupportedException* si la clase no implementa la interfaz Cloneable.
La interface Cloneable es muy simple (no define ninguna función):

```
public interface Cloneable {  }
```

DUPLICACIÓN DE UN OBJETO SIMPLE

La clase base Object de todas las clases en Java, tiene un método clone, que se redefine en la clase derivada para realizar una duplicación de un objeto de dicha clase.

Dada la clase Punto, para hacer una copia de un objeto de esta clase, se ha de:
1) implementar el interface Cloneable
2) redefinir la función miembro clone de la clase base Object

```
public class Punto implements Cloneable{
   private int x,y;
   ...
   public Object clone(){
      Object obj=null;
      try{
         obj=super.clone();              //se llama al clone( ) de la clase base (Object)
      }catch(CloneNotSupportedException ex){      //que hace copia binaria de x, y
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
   public String toString() { return origen+" ancho: "+ancho+" alto: "+alto;   }
}
```

En la redefinición de clone, la llamada a super.clone( ) se ha de hacer forzosamente dentro de un bloque try... catch, para capturar la excepción CloneNotSuportedException que nunca se producirá si la clase implementa el interface Cloneable.

```
      Punto punto=new Punto(20, 30);
      Punto pCopia=(Punto)punto.clone();
```

La promoción (casting) es necesaria ya que clone devuelve un objeto de la clase base Object que ha de ser promocionado a la clase Punto.

DUPLICACIÓN DE UN OBJETO COMPUESTO

Cuando un objeto contiene atributos que son objetos de otra clase, en la redefinición de la función miembro clone de la clase Contenedora se ha de efectuar una duplicación de dichos atributos subobjetos llamando a la versión clone definida en dichos subobjetos. De no hacerlo, los atributos subobjetos apuntarán al mismo objeto en el original y copia.

Ej Un objeto de la clase Rectangulo contiene un subobjeto de la clase Punto. En la redefinición de la función miembro clone de la clase Rectangulo se ha de efectuar una duplicación de dicho subobjeto llamando a la versión clone definida en la clase Punto.

Recuérdese que la llamada a clone siempre devuelve un objeto de la clase base Object que ha de ser promocionado (casting) a la clase derivada adecuada.

```
public class Rectangulo implements Cloneable{
   private int ancho ;
   private int alto ;
   private Punto origen;
//los constructores
   public Object clone(){
      Rectangulo obj=null;
      try{
         obj=(Rectangulo)super.clone();  //devuelve un objeto que tiene una copia
                                         //binaria de los atributos ancho, alto, origen
      }catch(CloneNotSupportedException ex){   //obj.origen = = origen (apuntan a lo
         System.out.println(" no se puede duplicar");                    //mismo)
      }
      obj.origen=(Punto)obj.origen.clone();  //devuelve un objeto Punto que tiene una
                  // copia binaria de x, y.    Si se omite entonces obj.origen==this.origen
      return obj;
   }
   public String toString() {
      String texto=origen+" ancho: "+ancho+" alto: "+alto;
      return texto;
   }
 //otras funciones miembro
}
```

En la redefinición de clone, la llamada a super.clone( ) se ha de hacer forzosamente dentro de un bloque try... catch, para capturar la excepción CloneNotSuportedException que nunca se producirá si la clase implementa el interface Cloneable.

```
      Rectangulo rect=new Rectangulo(new Punto(0, 0), 4, 5);
      Rectangulo rCopia=(Rectangulo)rect.clone();
```

La promoción (casting) es necesaria ya que clone devuelve un objeto de la clase base Object que ha de ser promocionado a la clase Punto.

EJEMPLO COMPLETO DE DUPLICACIÓN DE OBJETOS Y DE OBJETOS COMPUESTOS:

```
package clonico;
public class Punto implements Cloneable{
   private int x, y;
   public Punto(int x, int y)  {  this.x = x;   this.y = y;    }
   public Punto() {  x=0;   y=0;    }
   public Object clone(){
      Object obj=null;
      try{
         obj=super.clone();
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      return obj;
   }
   public void trasladar(int dx, int dy) { x+=dx;  y+=dy;  }
   public String toString(){
      String texto="origen: ("+x+", "+y+")";
      return texto;
   }
}
```

```
package clonico;
public class Rectangulo implements Cloneable{
   private int ancho ;
   private int alto ;
   private Punto origen;
   public Rectangulo() {
      origen = new Punto(0, 0);
      ancho=0;    alto=0;
   }
   public Rectangulo(Punto p) {    this(p, 0, 0);    }
   public Rectangulo(int w, int h) {
            this(new Punto(0, 0), w, h);
   }
   public Rectangulo(Punto p, int w, int h) {
      origen = p;
      ancho = w;    alto = h;
   }
   public Object clone(){
      Rectangulo obj=null;
      try{
         obj=(Rectangulo)super.clone();
      }catch(CloneNotSupportedException ex){
         System.out.println(" no se puede duplicar");
      }
      obj.origen=(Punto)this.origen.clone();
      return obj;
   }
   public void mover(int dx, int dy) {
      origen.trasladar(dx, dy);
   }
   public int area() {    return ancho * alto;    }
   public String toString(){
      String texto=origen+" ancho: "+ancho+" alto: "+alto;
      return texto;
   }
}
```

```
package clonico;

public class ClonicoApp {
   public static void main(String[] args) {
      Punto punto=new Punto(20, 30);
      Punto pCopia=(Punto)punto.clone();
      System.out.println("punto "+ punto);
      System.out.println("copia "+ pCopia);

      Rectangulo rect=new Rectangulo(new Punto(1, 1), 4, 5);
      Rectangulo rCopia=(Rectangulo)rect.clone();
      System.out.println("rectangulo "+ rect);
      System.out.println("copia "+ rCopia);

      rect.mover(3, 3);
      System.out.println("rectangulo "+ rect);
      System.out.println("copia "+ rCopia);

      try  {        //espera a que pulse una tecla + INTRO
         System.in.read();
      }catch (Exception e) {  }
   }
}
```

Salida:

```
punto origen: (20, 30)
copia origen: (20, 30)
rectangulo origen: (1, 1) ancho: 4 alto: 5
copia origen: (1, 1) ancho: 4 alto: 5
rectangulo origen: (4, 4) ancho: 4 alto: 5
copia origen: (1, 1) ancho: 4 alto: 5
```

Si se omite lo gris la salida sería:

```
punto origen: (20, 30)
copia origen: (20, 30)
rectangulo origen: (1, 1) ancho: 4 alto: 5
copia origen: (1, 1) ancho: 4 alto: 5
rectangulo origen: (4, 4) ancho: 4 alto: 5
copia origen: (4, 4) ancho: 4 alto: 5
```

ya que la copia y el original apuntan al mismo objeto punto