TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ENSENADA

PROGRAMA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

REPORTE

# TEMA: COMPILADOR UTILIZANDO ANTLR Y JAVAFX

*Autores:*
Mata Araujo Kevin
No Control: 18760440

*Profesora:*
ME Xenia Padilla Madrid

**Lenguajes y Autómatas 2(enero-junio 2021)**

Ensenada B.C. México
10 de junio de 2021

# 1. Introducción

Este programa tiene el fin de que conforme se vaya actualizando este tome forma de un compilador en el cual uno podrá programar cosas básicas que solo requieran el uso de símbolos lógicos y el uso de las diversas estructuras if-else . este programa esta hecho con antlr y javaFX.

# 2. Java FX

JavaFX es un conjunto de paquetes de gráficos y medios que permite a los desarrolladores diseñar, crear, probar, depurar e implementar aplicaciones de cliente enriquecido que operan de forma consciente en diversas plataformas.

## 2.1 Interfaz Grafica

La interfaz gráfica echa para el programa fue totalmente construida con JavaFX gracias a la aplicación scene builder de javafx permite crear interfaces gráficas de manera muy sencilla, en este caso (Figura 1) se mejoro la estética de la interfaz añadiendo color, estilo a los botones y una barra de menú arriba junto con un icono personalizado.

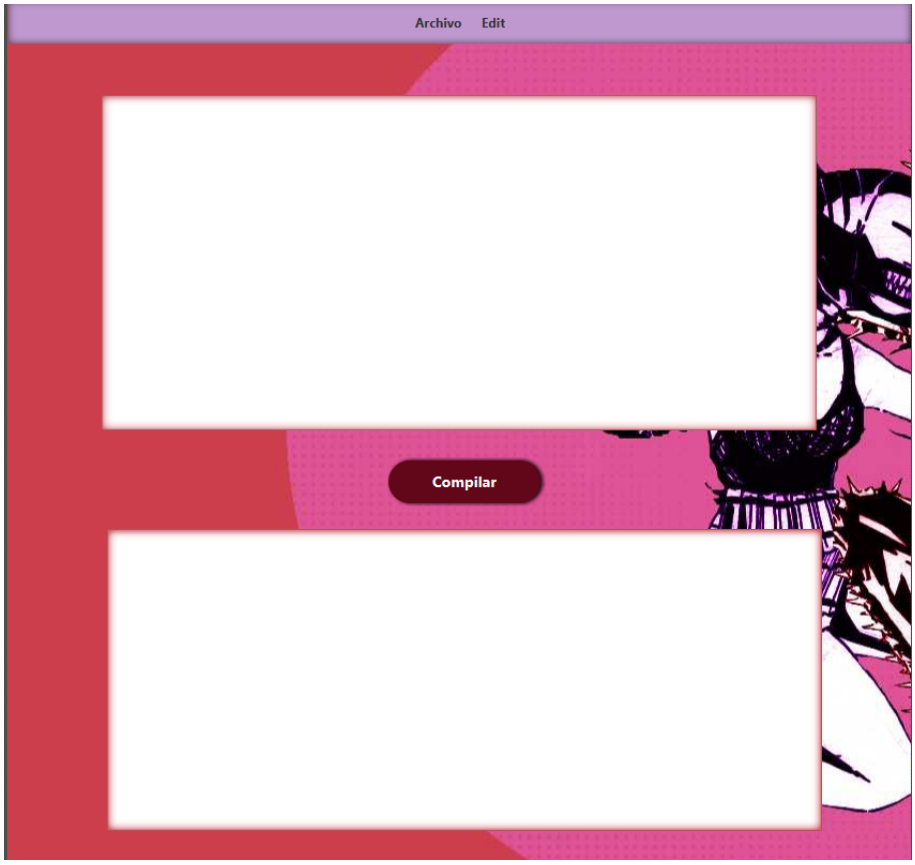



Figura 1: Interfaz Grafica

## 3. ANTLR

ANTLR cae dentro de la categoría de meta-programas, por ser un programa que escribe otros programas. Partiendo de la descripción formal de la gramática de un lenguaje, ANTLR genera un programa que determina si una sentencia o palabra pertenece a dicho lenguaje (reconocedor), utilizando algoritmos LL(*) de parsing. Si a dicha gramática, se le añaden acciones escritas en un lenguaje de programación, el reconocedor se transforma en un traductor o intérprete.

Además, ANTLR proporciona facilidades para la creación de estructuras intermedias de análisis (como ser ASTs - Abstract Syntax Tree), para recorrer dichas estructuras, y provee mecanismos para recuperarse automáticamente de errores y realizar reportes de los mismos.

ANTLR es un proyecto bajo licencia BSD, viniendo con todo el código fuente disponible, y preparado para su instalación bajo plataformas Linux, Windows y Mac OS X.

La gramática hecha para el programa cuenta con recursividad para que no haya problemas de jerarquía en las operaciones matemáticas y así no importe el tamaño de esta, el árbol binario que se crea a partir de la gramática siempre va a tener un sentido lógico. En esta nueva version la gramática fue configurada en varias capaz para poder tener un código mas optimizado. (Figura 2)

```
start:
          GALLETA MAIN PAR_A PAR_C CA inicio+ CC
        ;
inicio:
    declaraciones| asignaciones|impresion| impresioncadena|ifs| espacios| mientras ;
impresion :
        IMPRIMIR PAR_A expr PAR_C PC NUEVALINEA       #impresionExpresion ;
declaraciones :
        TIPO ID PC NUEVALINEA #declaracion ;
asignaciones:
        ID IGUAL expr PC NUEVALINEA #asigancion ;
espacios:
        NUEVALINEA               #espacio;
impresioncadena :
        IMPRIMIR PAR_A CADENA PAR_C PC NUEVALINEA  ;
ifs:
        CAMARA PAR_A condiciones_Logicas  PAR_C CA inicio*? CC  sinocondicion* sino?  #Condicion;
sino:
ESQUINA CA inicio+ CC #elsefinal
;
sinocondicion:
 ESQUINA CAMARA PAR_A condiciones_Logicas PAR_C CA inicio+ CC sino? #elsecondicion
    ;
mientras:
    MIENTRAS PAR_A simbolos PAR_C CA inicio*? CC
    ;
```

```
simbolos:
      expr  DOBLEIG expr #igual
       |
      expr MAYORK expr #Mayorq
       |
      expr MENORK expr #Menorq
       |
      expr  MAYORKIG expr   #Mayorqig
       |
      expr  MENORKIG expr   #Menorqig
       |
      expr DIFERENTE expr  #Diferentepa
       |
      expr     #verdad
       |
      PAR_A expr OR expr PAR_C #exprOR
       |
      PAR_A expr AND expr PAR_C #exprAND
       |
      expr OR expr AND expr #exprMultiple
       ;
condiciones_Logicas:
            simbolos  (AND  simbolos  )*? #and
           |
            simbolos  (OR  simbolos )*? #or
            |
            simbolos #simb
           ;
expr:
        expr op= (POR|DIV) expr     #MulDIv
        |
        expr op= (SUMA | RESTA) expr    #SumRes
        |
        INT                         #int
        |
        ID                          #id
        |
        PAR_A expr PAR_C             #parentesis
        ;
```

Figura 2: Gramática

### 3.1 Tokens

un token es un “tipo” (un entero) un texto (una cadena) y una línea y columna (sendos Enteros)”, el valor y nombre de estos son puestos por el programador y la forma en que los manipula depende de la gramática. En la figura 3 se pueden observar los tokens que se crearon para nuestra gramática.

```
GALLETA: 'galleta';
MAIN: 'main' ;
TIPO: 'int' ;
IMPRIMIR:'imprimir';
CAMARA:'camara';
ESQUINA:'esquina';
 ID:[a-z][a-zA-Z0-9]+ ;
 IGUAL: '=';
 DOBLEIG:'==';
 MAYORK:'>';
 MAYORKIG:'>=';
 MENORK:'<';
 MENORKIG:'<=';
 DIFERENTE: '!=';
CADENA:'"' [ \t\r\na-zA-Z0-9]*?'"' ;
POR:'*';
DIV:'/';
SUMA:'+';
RESTA:'-';
INT:[0-9]+;
CA: '[';
CC:']';
PAR_A:'(';
 PAR_C:')';
NUEVALINEA:[\r\n]+;
ESPACIOS: [ \t]+ ->skip;
FLOTANTE: 'flotante';
NUM_FLOTANTE: [0-9]+'.'[0-9]+;
LINE_COMMENT: '//' .*? '\n' ->skip;
COMENTARIO: '/*' .*? '*/'-> skip;
```

Figura 3: Tokens

## 4. Funcionamiento del programa

Para poder darle un procesos lógico a las operaciones creamos la clase java MyVisitor donde se almacenan y programan todas las visitas a las diferentes capas de la gramática, a continuación están las diferentes funciones que se crearon para cada parte de la gramática

```
@Override public Integer visitInt(OperacionesParser.IntContext ctx)
{
/*
*convertimos el lexema a numnero
 */
    return Integer.valueOf(ctx.INT().getText());
}
@Override public Integer visitId(OperacionesParser.IdContext ctx)
{
    String id = ctx.ID().getText();
    if(memoria.containsKey(id)){
        return memoria.get(id) ;
    }else{
        if(errores == 0 ){
            resultado += "\nLa variable " + id + " No existe ";
            errores++;
        }
    }
    return memoria.get(ctx.ID().getText()) ;
}
```

Figura 4: conversión de lexemas

como se puede observar en la figura 4 dentro de la visita que ocurre a a las partes de la gramática llamada id y int podemos observar como aquí convertimos el lexema a un numero y el lexema de id se almacena en el hashmap memoria para poder almacenar variables

```java
@Override public Integer visitSumRes(OperacionesParser.SumResContext ctx)
{
    int izq = visit(ctx.expr( i: 0));
    int der = visit(ctx.expr( i: 1));
    if(ctx.op.getType() == OperacionesParser.SUMA){

        return izq +der;
    }
    return izq - der;
}
@Override public Integer visitImpresionExpresion(OperacionesParser.ImpresionExpresionContext ctx)
{
    Integer valor = visit(ctx.expr());
    if(errores == 0 ) {

        String res = "\nEl resultado es " + valor;
        resultado += res;
    }

    return valor;
}
@Override public Integer visitMulDIv(OperacionesParser.MulDIvContext ctx)
{
    int izq = visit(ctx.expr( i: 0));
    int der = visit(ctx.expr( i: 1));

    if(ctx.op.getType() == OperacionesParser.DIV){
        if (der == 0 ){
                if(errores == 0){
                    resultado +="\n no se puede dividir entre cero viejo tonto";
                    errores++;
                }
            return 0;
        }
        return izq / der;
    }
    return izq * der;
}
```

Figura 5: Funciones de suma, resta, multiplicación, división y impresión de resultado

En la figura 5 se puede observar las funciones que devuelven las operaciones matemáticas esto se logra ya que se toma los hijos de izquierda y derecha y ya lo único que hace es realizar la operación, en el caso de la impresión el valor final se almacena en la variable valor la cual recibe la expresión final, al mismo se crea una variable llamada resultado donde se almacena el valor para poder imprimirlo en la consola de la aplicación.

```java
@Override public Integer visitAsigancion(OperacionesParser.AsigancionContext ctx) {

    String id = ctx.ID().getText();
    int valor = visit(ctx.expr());
    if( memoria.containsKey(id)){
        memoria.put(id,valor);
    }else{
        /*throw new ExceptionErrordedeclaracion(id);*/
        if (errores == 0 ) {
            resultado += "\nLa variable "  +id + " No esta delcarada";
            errores++;
        }
    }
    return valor;
}
```

Figura 6: Asignación y Paréntesis

para finalizar en la figura 6 se puede observar la asignación de valores para una variable que se requiera usar y la la función de paréntesis la cual solo asignara el contenido de expresión en el caso de que este se contenga en unos paréntesis.

## 4.1 if-else-else-if

En esta nueva actualización al programa, se le agrego la función de if-else la cual le permite crear condiciones y con estas poder crear programas mas complejos, en este caso como esta programado mi gramática me permite hacer if, if-else,else if y if anidados, la programación en esta gramática se hizo en cada apartado del símbolo lógico necesario (==,!=,¿,etc) a continuación esta es la programación echa en la clase MyVisitor. (Figura 7 )

```
@Override public Integer visitIgual(OperacionesParser.IgualContext ctx)
{
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 == exp2 ){
        return 1;
    }
    return 0;
}
@Override public Integer visitMayorq(OperacionesParser.MayorqContext ctx)
{
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 > exp2 ){
        return 1;
    }
    return 0;
}
```

```
@Override public Integer visitMenorq(OperacionesParser.MenorqContext ctx) {
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 < exp2 ){
        return 1;
    }
    return 0;
}
@Override public Integer visitMayorqig(OperacionesParser.MayorqigContext ctx)
{
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 >= exp2 ){
        return 1;
    }
    return 0;
}
```

```
@Override public Integer visitMenorqig(OperacionesParser.MenorqigContext ctx)
{
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 <= exp2 ){
        return 1;
    }
    return 0;
}

@Override public Integer visitDiferentepa(OperacionesParser.DiferentepaContext ctx)
{
    int exp1 = visit(ctx.expr( 0));
    int exp2 = visit(ctx.expr( 1));
    if (exp1 != exp2 ){
        return 1;
    }
    return 0;
}
```

Figura 7: Configuración de los operadores

los códigos anteriores son solo la configuración de los operadores lógicos a continuación se encuentra la programación para if en este caso condición, y el else en este caso elsefinal seria el else normal sin condiciones y elsecondicion seria else-if, este programa permite el saber la cantidad de código dentro de cada una de las condiciones. (Figura 8)

```java
/*este es el if basico no seas wilo*/
@Override public Integer visitCondicion(OperacionesParser.CondicionContext ctx)
{
int k = 0 ;
int m = 0;
    if (visit(ctx.condiciones_Logicas())==1){
        while (ctx.inicio(k) != null) {
            visit(ctx.inicio(k));
            k++;
        }
        return  1 ;
    }else {
        while(ctx.sinocondicion(m) != null){
            visit(ctx.sinocondicion(m));
            m++;
        }
        if(ctx.sino() != null){
            visit(ctx.sino());
        }
    }
return 0;

}
```

```java
@Override public Integer visitElsefinal(OperacionesParser.ElsefinalContext ctx) {
 return visitChildren(ctx);
}

@Override public Integer visitElsecondicion(OperacionesParser.ElsecondicionContext ctx)
{
int i = 0 ;
if (visit(ctx.condiciones_Logicas())==1){
    while (ctx.inicio(i) != null) {
        visit(ctx.inicio(i));
        i++;
    }
    return  1 ;
}else {
    if (visit(ctx.sino()) != null){
        visit(ctx.sino());
    }
    return  0 ;
}
}
```

Figura 8: Configuración de if-else else-if

En el uso de if-else se pueden usar operadores lógicos para hacer un código mas optimizado y condiciones mas complejas en este caso programamos los operadores lógicos or y and.(Figura 9)

```java
@Override public Integer visitAnd(OperacionesParser.AndContext ctx)
{
    int i = 0;
    while(ctx.simbolos(i) !=null){
        if(visit(ctx.simbolos(i)) ==0 ){
            return 0;
        }
        i++;
    }
    return 1;
}
@Override public Integer visitOr(OperacionesParser.OrContext ctx)
{
    int i = 0;
    while(ctx.simbolos(i) !=null){
        if(visit(ctx.simbolos(i)) ==1 ){
            return 1;
        }
        i++;
    }
    return 0;
}
```

Figura 9: Operadores lógicos

## 4.2 While

Como se puede ver en la gramática actualizada en esta version, añadimos ciclo while, uno de los ciclos mas usados el cual básicamente hace funcionar instrucciones mientras una condición este activa y sea verdadera a continuación esta la programación de este ciclo. (Figura 10)

```
@Override public Integer visitMientras(OperacionesParser.MientrasContext ctx)
{
    int validacion = visit(ctx.simbolos());
    while(validacion ==1){
        visitChildren(ctx);
        validacion = visit(ctx.simbolos());
    }
    return 0;
}


}
```

Figura 10: While

## 4.3 Variables Flotantes

Para poder hacer uso de variables flotantes lo que se hizo fue crear un hashmap especifico que pueda almacenar variables flotantes y otro para variables enteras, asi que los procesos que se hicieron para las variables enteras se hacen por separado siguiendo los mismos pasos pero esta vez se almacenan en su hashamap determinado como se puede ovbservar en la siguiente figura

```
Map<String,Float> memoria = new HashMap<~>();
Map<String,Float> memoriaFlotantes = new HashMap<~>();
Map<String,Float> memoriaTemp = new HashMap<~>();
Map<String,Float> memoriaFlotantesTemp = new HashMap<String,Float>();
```

```
@Override public Float visitNumFlotante(OperacionesParser.NumFlotanteContext ctx) {
    return Float.parseFloat(ctx.NUM_FLOTANTE().getText());
}
```

```
@Override public Float visitAsignacion(OperacionesParser.AsignacionContext ctx) {
    String id = ctx.ID().getText();
    if(errores == 0){
        if(memoria.containsKey(id)){
            Float valor = visit(ctx.expr());
            memoria.put(id,valor);
            return valor;
        } else if(memoriaFlotantes.containsKey(id)){
            Float valor = visit(ctx.expr());
            memoriaFlotantes.put(id,valor);
            return valor;
        }
        else {
            if(errores == 0){
                resultado += "\nErrore de variable " + id + "' no a sido declarada en el codigo";
                errores++;
            }

        }
    }
```

Figura 11: Funciones para variables flotantes

como se puede observar en las figuras anteriores la programación nueva que se da es la recuperación del contenido de la variables y la implementacion de esta en su respectivo hashmap dependiendo de si es flotante o entero.

## 4.4 Scope de variables

El scope de variable se refiere a la jerarquía de bloques que existe en un programa y el como esta debe ser respetada ya que dependiendo de donde se encuentre una variables puede que esta tenga la posibilidad de ser usada o no, por ejemplo en el caso de un if en un programa si la condición no se cumple, las variables y modificaciones dentro del if se verán ignoradas por el programa y nada fuera del if tendrá acceso a su contenido.

Para tener el scope de variables en nuestro programa se creo un subespacio en la gramática (Figura 12 ) donde se mande a llamar todo el contenido de esta, con el fin de poder programar en ese espacio, como se pudo observar en la figura 11 se crearon no solo hashamaps para el uso de enteros y flotantes sino que ademas se creo un hashamap extra de cada uno que usaremos para almacenar datos temporalmente, entonces el programa funcionara de la siguiente manera cada vez que se adentre en un bloque se copiara toda la información de los hashmaps principales a los temporales y son estos los que se utilizaran para las funciones dentro de del dicho bloque al que entramos.

```
cuerpo2: cuerpo*;
```

```
@Override public Float visitCuerpo2(OperacionesParser.Cuerpo2Context ctx) {
    memoriaTemp = new HashMap<String,Float>(memoria);
    memoriaFlotantesTemp = new HashMap<String,Float>(memoriaFlotantes);
    visitChildren(ctx);
    return 0f;
}
```

Figura 12: hashmaps temporales

Después se hizo una función que se manda a llamar después de cada bloque que usemos la cual se encargara de vaciar las variables temporales, esto se encarga de que la memoria no se llene y se respete la jerarquía de bloques (Figura 13)

```
rivate void actualizarHash(){
   Iterator<Map.Entry<String,Float>> memoriaIterator = memoria.entrySet().iterator();
   Iterator<Map.Entry<String,Float>> memoriaIterator2 = memoriaFlotantes.entrySet().iterator();

   while(memoriaIterator.hasNext() || memoriaIterator2.hasNext()){
       Map.Entry<String , Float> entry = memoriaIterator.next();
       if(!memoriaTemp.containsKey(entry.getKey())){
           memoria.remove(entry.getKey());
       }else if (!memoriaFlotantesTemp.containsKey(entry.getKey())){
           memoriaFlotantes.remove(entry.getKey());
       }
   }
   memoria = new HashMap<String,Float>(memoriaTemp);
   memoriaTemp.clear();
```

Figura 13: Limpieza de variables temporales

# 5. Programación de los botones

## 5.1. Programación Botón Imprimir/Compilar

En lo que se refiere a la programación del botón para imprimir o calcular las operaciones, en este se usan las funciones File, FileWriter, BufferedWriter y Printwriter, estas se usan para que el contenido en el text área se guarde en un documento txt y este pueda ser leído por lo siguiente el cual es la declaración y uso de las operación del léxico, tokens y sintáctico, esta partees para poder hacer uso de la gramática. Al final solo se crea una instancia de la clase MyVisitor y se usa para visitar el árbol generado. (Figura 14)

```
public void pressbutton(ActionEvent event) throws IOException {

    File f;
    FileWriter w;
    BufferedWriter bw;
    PrintWriter wf;

    try {
        f = new File( pathname: "data.txt");
         w = new FileWriter(f);
        w.write(textareachido.getText());
        w.close();
        bw= new BufferedWriter(w);
        wf = new PrintWriter(bw);

    }catch (Exception a){
        System.out.println("no jala tu chingadera ");
    }
    CharStream input = CharStreams.fromFileName("data.txt");
    OperacionesLexer lexico = new OperacionesLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexico);
    OperacionesParser sintactico = new OperacionesParser(tokens);
    ParseTree arbol = sintactico.start();

    MyVisitor visitas =new MyVisitor();
    visitas.visit(arbol);
    areasalida.appendText(visitas.getResultado());
}
```

Figura 14: Programación del botón imprimir

## 5.2. Programación Botón Archivo.

En esta mejora del programa se añadió una tool bar en donde se encuentran los botones archivo y clear en este caso el botón archivo fue programado para poder abrir documentos de texto en el text área de entrada en el programa. (Figura 15)

## 5.3. Programación Botón Clear

En este botón solo se programo el comando para limpiar el text área de salida que se esta utilizando como consola. (Figura 16)

```java
public void archivo(ActionEvent event)throws IOException{
    String aux = "";
    String texto = "";
    String resultado = "";
    try {
        JFileChooser file = new JFileChooser(System.getProperty("user.dir"));
        file.showOpenDialog( parent: null);
        File archivo = file.getSelectedFile();
        if (archivo != null) {
            FileReader archivos = new FileReader(archivo);
            BufferedReader leer = new BufferedReader(archivos);
            while ((aux = leer.readLine()) != null) {
                texto += aux + "\n";
            }
            leer.close();
        }
    } catch (IOException ex) {
        JOptionPane.showMessageDialog( parentComponent: null, message: "Error Importando - " + ex);
    }
    textareachido.setText(texto);
}
```

Figura 15: Programación del botón archivo

```java
public void borrar (ActionEvent event)throws IOException{
    areasalida.clear();
}
```

Figura 16: Programación Botón Clear

## 5.4. Programación del Botón traducir

Este botón se encarga de traducir el idioma de programación c al el lenguaje creado por sus servidor, funciona gracias principalmente a el código replaceAll que permite el cambiar un texto especifico por otro.(Figura 17)

```java
public void traductor(ActionEvent event)throws  IOException{
    FileWriter fw = new FileWriter( fileName: "C:\\Users\\COOLER MASTER\\Desktop\\automatas2\\Calculadora\\data.txt");
    fw.write(textareachido.getText());
    fw.close();

    String texto = "";
    FileReader entrada = new FileReader( fileName: "C:\\Users\\COOLER MASTER\\Desktop\\automatas2\\Calculadora\\data.txt");
    int c = 0;
    while(c != -1){
        c = entrada.read();
        char letra = (char)c;
        texto += letra;
    }
    entrada.close();
    String texto2 = texto.replaceAll( regex: "void", replacement: "galleta");
    String texto3 = texto2.replaceAll( regex: "\tint", replacement: "int");
    String texto4 = texto3.replaceAll( regex: "printf", replacement: "imprimir");
    String texto5 = texto4.replaceAll( regex: "if", replacement: "camara");
    String texto6 = texto5.replaceAll( regex: "else", replacement: "esquina");
    String texto7 = texto6.replaceAll( regex: "#include <stdio.h>", replacement: " ");
    String texto8 = texto7.replaceAll( regex: ";", replacement: " ");
    String texto9 = texto8.replaceAll( regex: "'(  ')'", replacement: "");
    textareachido.clear();
    texto9 = texto9.substring(0, texto9.length() -1);
    textareachido.appendText(texto9);
}
```

Figura 17: Botón Traductor

## 5.5. Programacion del boton traductor inverso

Ahora este botón lo que va a ser es convertir nuestro lenguaje personalizado a lenguaje C, utiliza las mismas funciones de el botón traductor normal. (Figura 18)

```
public void traductorc(ActionEvent event )throws  IOException{

    FileWriter fw = new FileWriter( fileName: "C:\\Users\\COOLER MASTER\\Desktop\\automatas2\\Calculadora\\data.txt");
    fw.write(textareachido.getText());
    fw.close();

    String texto = "#include <stdio.h>";
    FileReader entrada = new FileReader( fileName: "C:\\Users\\COOLER MASTER\\Desktop\\automatas2\\Calculadora\\data.txt");
    int c = 0;
    while(c != -1){
        c = entrada.read();
        char letra = (char)c;
        texto += letra;
    }
    entrada.close();
    String texto2 = texto.replaceAll( regex: "galleta",  replacement: "void");
    String texto3 = texto2.replaceAll( regex: "int",  replacement: "\tint");
    String texto4 = texto3.replaceAll( regex: "imprimir",  replacement: "printf");
    String texto5 = texto4.replaceAll( regex: "camara",  replacement: "if");
    String texto6 = texto5.replaceAll( regex: "esquina",  replacement: "else");
    String texto7 = texto6.replaceAll( regex: "mientras", replacement: "while");
    String texto8 = texto7.replaceAll( regex: "'(' ')'",  replacement: "");
    textareachido.clear();
    texto8 = texto8.substring(0, texto8.length() -1);
    textareachido.appendText(texto8);
}
```

Figura 18: Botón Traductor Inverso

## 6. Salida

Para finalizar este es el como queda el programa al final (Figura 19) se puede observar su funcionamiento utilizando un ejemplo básico.

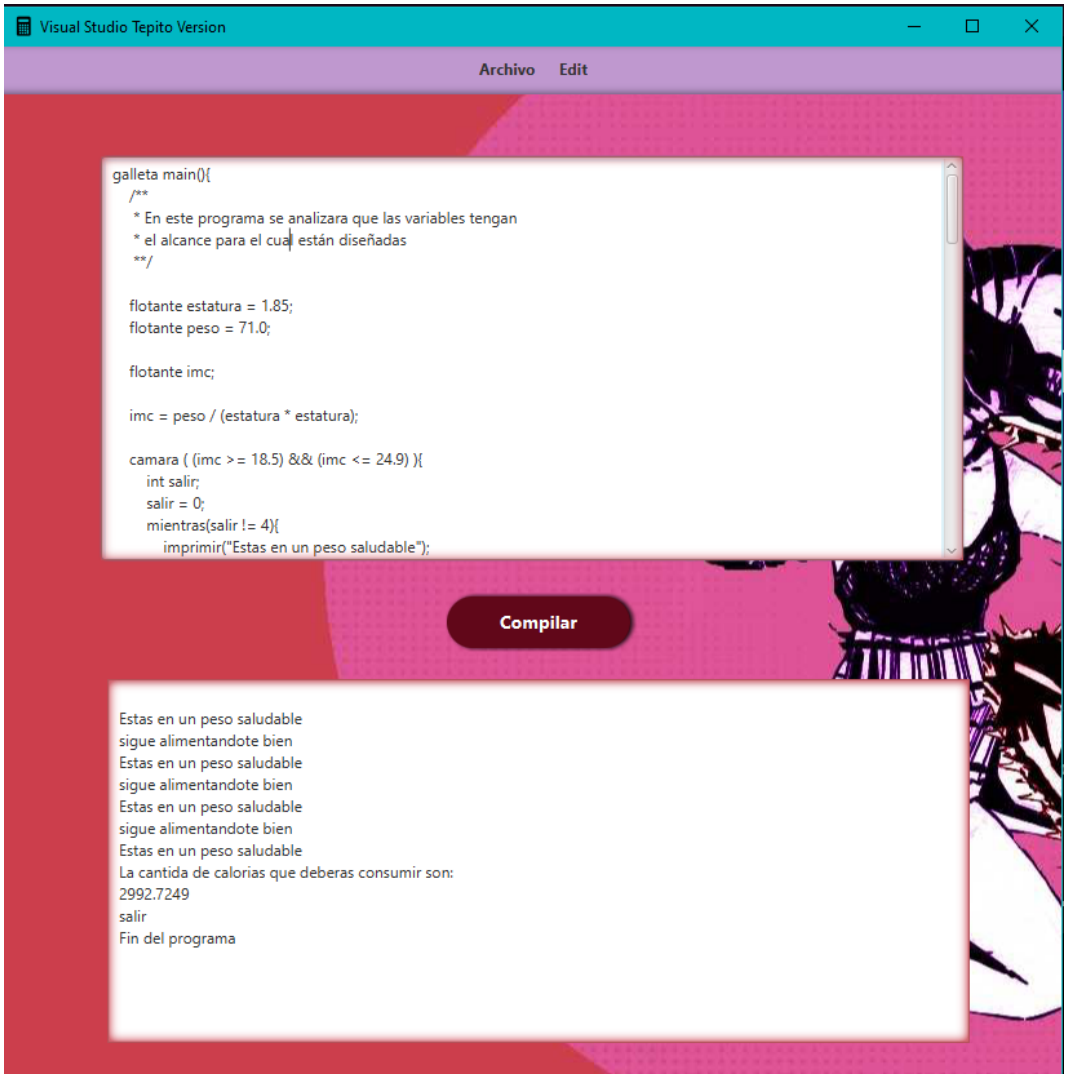
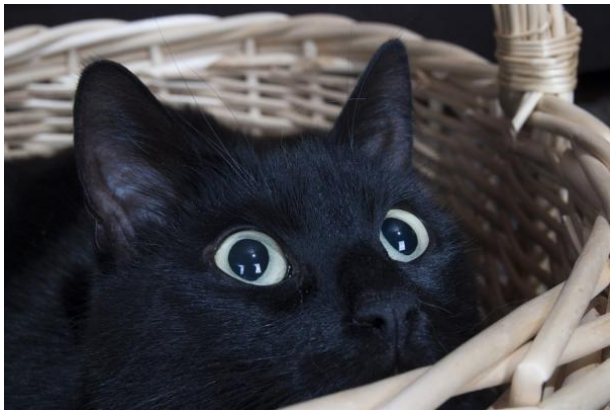



Figura 19: Salida

## 7. Conclusión

Después de llevar ya un tiempo utilizando antlr la verdad es que me parece una herramienta muy útil y eficaz ya que si la gramática esta bien hecha se pueden llegar a resultados óptimos, a pesar de que odie el antlr ya que se me complico mucho creo que debería de estudiarlo mas a fondo para poder utilizarlo en futuros proyectos que requieran un nivel alto de programación, y ya son mas qque decir proseroma muchas gracias por enseñarnos su clase es de las mejores y en donde mas e aprendido espero que en un futuro cercano me pueda volver a dar clases en otras materias .



Gracias por su atención