

# Índice

## 1. Introducción

- 1.1 Consideraciones generales sobre los prototipos
- 1.2 Gramática a utilizar en el proyecto
- 1.3 Detalles del desarrollo del proyecto

## 2. Configuración, ejecución y fichero de prueba

- 2.1 Configuración de nuestro sistema
- 2.2 Ejecución del compilador
- 2.3 Fichero de prueba

## 3. Explicación de los ficheros del compilador

- 3.1 Analizador Morfológico
- 3.2 Analizador Sintactico

# 1. Introducción

En este proyecto hemos realizado la construcción de un compilador a través de la realización de prototipos, con ayuda de las herramientas léxico-semánticas como son CUP y jFlex y utilizando el lenguaje de programación Java SE.

El objetivo principal de esta práctica de Procesadores de Lenguaje es la construcción de un compilador para el lenguaje descrito posteriormente. El compilador realizado traduce un archivo que contiene código escrito en nuestro lenguaje a su equivalente ensamblador, obteniendo las instrucciones correspondientes a las acciones descritas en nuestro lenguaje en el código ensamblador.

El compilador se ha construido con un modelo de ciclo de vida incremental basado en prototipos, ya que en el enunciado de la práctica se definía así.

Los prototipos que se han seguido son los siguientes:

- Prototipo 0: Análisis inicial de la estructura de datos del compilador.
- Prototipo 1: Declaraciones de tipos (el tipo puntero y matriz se consideran opcionales). Se corresponde en la gramática con Bloque-Declaración
- Prototipo 2: Sentencias con expresiones: constantes, aritméticas y lógicas; además de entrada (lectura)/salida (escritura). Se corresponde en la gramática con Sentencias aritméticas, sentencias lógicas, sentencias entrada/salida
- Prototipo 3: Sentencia de bloque condicional.
- Prototipo 4: Sentencia de bloque bucle (iteración).
- Prototipo 5: Sentencias con vectores.
- Prototipo 6: Funciones.

En nuestra práctica sólo hemos realizado los prototipos cero y uno, debido a un ajuste de planificación.

## 1.1 Consideraciones generales sobre los prototipos

### 1.1.1 Gestión de errores léxicos, sintácticos y semánticos

Cuando un error léxico se produce, el analizador léxico muestra un error cuyo formato ha

sido definido de la siguiente manera:

— Error léxico

```
ERROR LEXICO: <nº linea> <nº caracter>:<codigo> <descripcion>
```

- <nº linea>: Línea donde aparece el token erróneo.
- <nº caracter>: Carácter en la correspondiente columna donde empieza el token erróneo.
- <codigo>: Valor numérico que indica el número de error.
- <descripcion>: Descripción del error.

Cuando se produce un error sintáctico el analizador mostrará un mensaje siguiendo el siguiente formato:

— Error sintáctico

```
ERROR SINTACTICO: <nº linea> <nº caracter>:<codigo> <descripcion>
```

- <nº linea>: Línea donde aparece el token erróneo.
- <nº caracter>: Carácter en la correspondiente columna donde empieza el token erróneo.
- <codigo>: Valor numérico que indica el número de error.
- <descripcion>: Descripción del error.

Cuando un error semántico se produce, siguiendo el mismo esquema que los demás errores, la salida del error seguirá el siguiente formato:

— Error semántico

```
ERROR SINTACTICO: <nº linea> <nº caracter>:<codigo> <descripcion>
```

- <nº linea>: Línea donde aparece el token erróneo.
- <nº caracter>: Carácter en la correspondiente columna donde empieza el token erróneo.
- <codigo>: Valor numérico que indica el número de error.
- <descripcion>: Descripción del error.

## 1.1.2 Análisis semántico

Se han considerado como válidos los aspectos semánticos utilizados en la mayor parte de los lenguajes de programación imperativos, tal y como dicta el enunciado.

## 1.1.3 Generación de código

El compilador traduce los programas descritos mediante la gramática propuesta a programas en lenguaje ensamblador. Esta generación de código se ha realizado mediante la asociación de acciones a la reducción de las reglas de la gramática descritas en el analizador sintáctico.

## 1.2 Gramática a utilizar en el proyecto

Para realizar el proyecto se ha llevado a cabo la implementación de la siguiente gramática para utilizar en nuestro compilador:

— Gramática del proyecto

### Descripción del programa Principal

```
<programa> ::= program { <Bloque_declaración> <funciones> <sentencias> }
<Bloque_declaracion> ::= <declaraciones_constante>
                        | <declaraciones>
                        | <declaraciones_constante><declaraciones>
```

### Apartado de declaraciones para el programa y las funciones

```
<declaraciones_constante> ::= <declaracion_constante>
                           | <declaracion_constante> <declaraciones_constante>
<declaracion_constante> ::= const <clase_escalor> <identificador>=<constante>
<declaraciones> ::= <declaracion>
                  | <declaracion> <declaraciones>
<declaracion> ::= <clase> <identificadores> ;
```

### Tipos de datos para el lenguaje

```
<clase> ::= <clase_escalor>
          | <clase_puntero>
          | <clase_vector>
          | <clase_matriz>
<clase_escalor> ::= <tipo>
<tipo> ::= int
        | boolean
        | float
<clase_puntero> ::= <tipo> *
                  | <clase_puntero> *
<clase_vector> ::= array[ <constante_entera> ] of <tipo>;
<clase_matriz> ::= array[<constante_entera>,<constante_entera>] of <tipo>;
<identificadores> ::= <identificador>
                   | <identificador> , <identificadores>
```

### Definición de funciones del lenguaje

```
<funciones> ::= <funcion> <funciones>
              | <funcion>
<funcion> ::= function <tipo> <identificador> ( <parametros_funcion> ) {
<declaraciones_funcion> <sentencias>}

<parametros_funcion> ::= <parametro_funcion> <resto_parametros_funcion>
<resto_parametros_funcion> ::= ; <parametro_funcion> <resto_parametros_funcion>
<parametro_funcion> ::= <tipo> <identificador>
<declaraciones_funcion> ::= <declaraciones>
                          | <declaraciones_constante>
```

Definición de sentencias del lenguaje

```
<sentencias> ::= <sentencia>
               | <sentencia> <sentencias>
<sentencia> ::= <sentencia_simple> ;
               | <bloque>
```

```

<sentencia_simple> ::= <asignacion>
                    | <lectura>
                    | <escritura>
                    | <liberacion>
                    | <retorno_funcion>
<bloque> ::= <condicional>
            | <bucle>

```

#### **Sentencia de asignación**

```

<asignacion> ::= <identificador> = <exp>
              | <elemento_vector> = <exp>
              | <acceso> = <exp>
              | <identificador> = malloc <identificador> = & <identificador>
<elemento_vector> ::= <identificador> [ <exp> ]
                  | <identificador> [ <exp> , <exp> ]

```

#### **Sentencia condicional**

```

<condicional> ::= if ( <exp> ) { <sentencias> }
               | if ( <exp> ) { <sentencias> } else { <sentencias> }

```

#### **Sentencia iterativa**

```

<bucle> ::= while ( <exp> ) { <sentencias> }
        | for ( <identificador> = <exp> ; <exp> ) { <sentencias> }

```

#### **Sentencia de lectura/escritura**

```

<lectura> ::= scanf <identificador>
           | scanf <elemento_vector>
<escritura> ::= printf <exp>
             | cprintf <identificador>

```

#### **Sentencias de punteros**

```

<liberacion> ::= free <identificador>
<acceso> ::= * <identificador>
          | * <acceso>

```

#### **Sentencias de retorno de función**

```

<retorno_funcion> ::= return <exp>

```

#### **Sentencias aritméticas, lógicas**

```

<exp> ::= <exp> + <exp>
        | <exp> - <exp>
        | <exp> / <exp>
        | <exp> * <exp>
        | - <exp>
        | <exp> && <exp>
        | <exp> || <exp>
        | ! <exp>
        | <identificador>
        | <constante>
        | ( <exp> )
        | ( <comparacion> )
        | <acceso>
        | <elemento_vector>

```

```

    | size ( <identificador> )
    | contains ( <exp> , <identificador> )
    | <identificador> ( <lista_expresiones> )
<lista_expresiones> ::= <exp> <resto_lista_expresiones>
<resto_lista_expresiones> ::= , <exp> <resto_lista_expresiones>

```

#### **Sentencias relacionales**

```

<comparacion> ::= <exp> == <exp>
                | <exp> != <exp>
                | <exp> <= <exp>
                | <exp> >= <exp>
                | <exp> < <exp>
                | <exp> > <exp>

```

#### **Constantes, identificadores de variables**

```

<constante> ::= <constante_logica>
                | <constante_entera>
                | <constante_real>
                | <constante_logica> ::= true | false
<constante_entera> ::= <constante_decimal>
                    | <constante_binaria>
                    | <constante_hexadecimal>
                    | <constante_octal>
<constante_decimal> ::= +<numero>
                    | -<numero>
<constante_binaria> ::= +<numero>b
                    | -<numero>b
<constante_hexadecimal> ::= +<numero>h
                    | -<numero>h
<constante_octal> ::= +<numero>o
                    | -<numero>o
<numero> ::= <digito>
            | <numero> <digito>
<constante_real> ::= <constante_entera> . <constante_entera>

<identificador> ::= <letra>
                  | <letra> <resto_identificador>
<resto_identificador> ::= <alfanumerico>
                       | <alfanumerico> <resto_identificador>
<alfanumerico> ::= <letra>
                  | <digito>
<letra> ::= a | b | ... | z | A | B | ... | Z
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numero b> ::= 0|1
<numero h> ::= <digito>|A|B|C|D|E|F
<numero o> ::= 0|1|2|3|4|5|6|7

```

## 1.3 Detalles del desarrollo del proyecto

Para abordar el desarrollo del proyecto, se ha partido de la gramática definida en el ‘Incremento 2’, donde ya implementamos la gramática descrita en el apartado 1.2 (proyecto de Jesús, Pablo y Luis), junto con la gestión de errores y funciones auxiliares integradas en el mismo incremento (proyecto de José). Con la unificación de dichos elementos, procedimos a salir del entorno de desarrollo hasta ahora empleado (Netbeans) para simplificar la unión y gestión de las clases Java. Empleamos la aproximación propuesta en la asignatura de Autómatas, donde realizamos la generación del Analizador morfológico (con la librería JFlex) y el Parser (con CUP) mediante un fichero de compilación automática .bat (1GenerarCompilador.bat), y posteriormente se compilará el fichero con el programa de prueba mediante otro .bat (2CompilarTexto.bat) que generará un fichero .txt con la salida correspondiente.

Respecto a la salida producida por nuestro compilador, estuvimos debatiendo entre generar código en el lenguaje ensamblador NASM o un código intermedio basado en tercetos/cuartetos. Dado que la mayoría de los miembros del grupo habían trabajado anteriormente con NASM, nos decidimos a abordar esta solución.

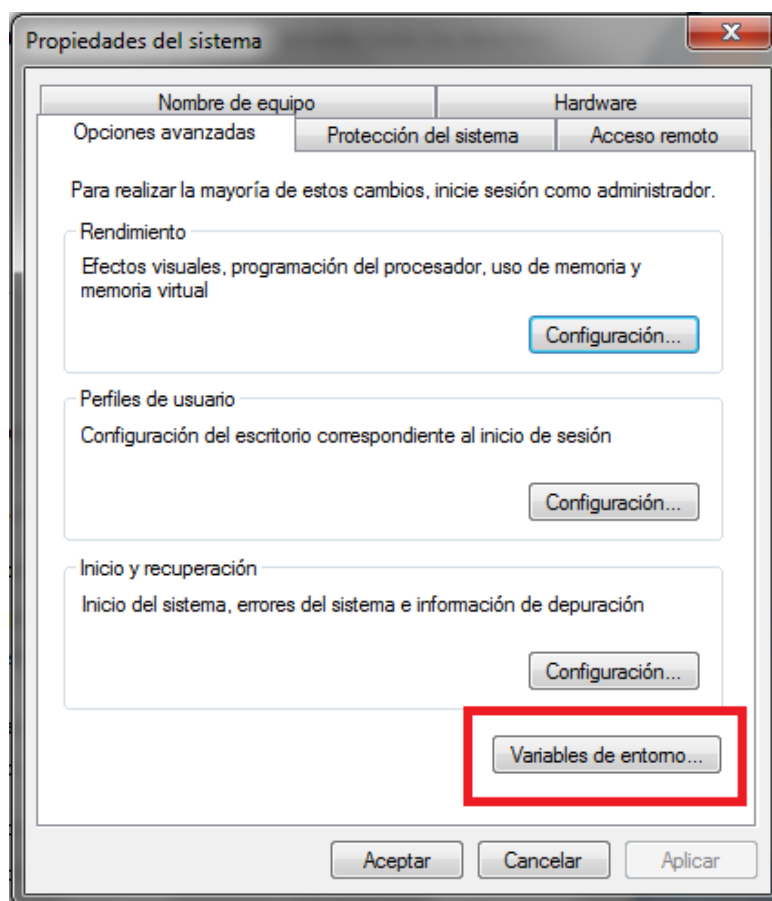
Queremos resaltar que tuvimos problemas con nuestra implementación de las comparaciones y la pila de los condicionales (IF), pues no teníamos muy claro el procedimiento de apilación y la secuencia NASM correspondiente, por tanto hemos realizado una implementación que creemos es correcta, pero no podemos asegurar que la salida producida óptima.

## 2. Configuración y ejecución

### 2.1 Configuración de nuestro sistema

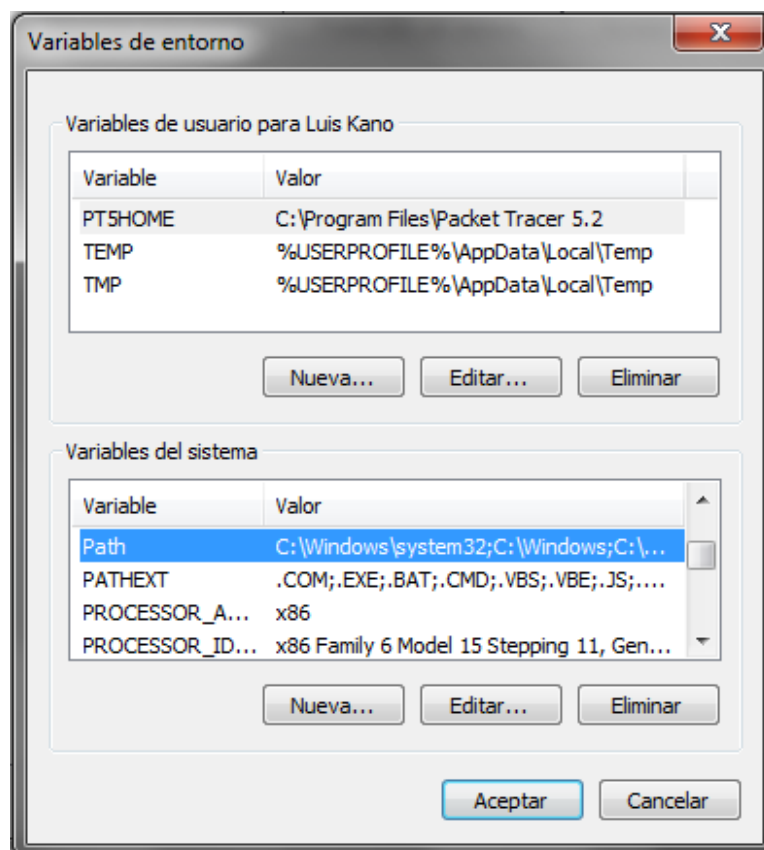
Para que el compilador funcione debemos instalar Java en nuestro ordenador y además hay que configurar las variables de entorno para que podamos “compilar” desde la Consola de Comandos de Windows 7, que es lo que necesitamos para que funcionen los scripts “.bat” que lanzan la compilación..

Damos Clic Derecho a Mi PC y nos vamos a Propiedades. Y nos aparecerá una ventana donde aparece la información de nuestro sistema, Damos clic en Configuración Avanzada del Sistema, que se encuentra a la izquierda. Y nos debe de aparecer una ventana como esta:

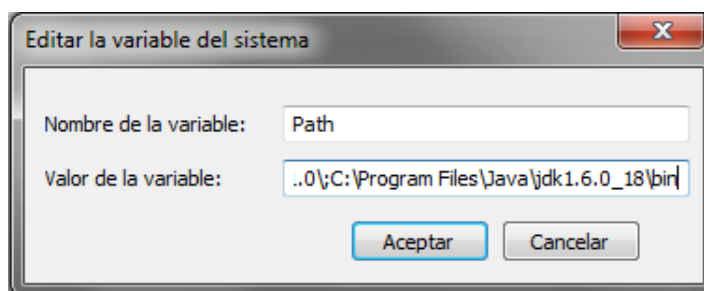


Nos vamos a la pestaña de Opciones Avanzadas y damos clic en Variables de Entorno



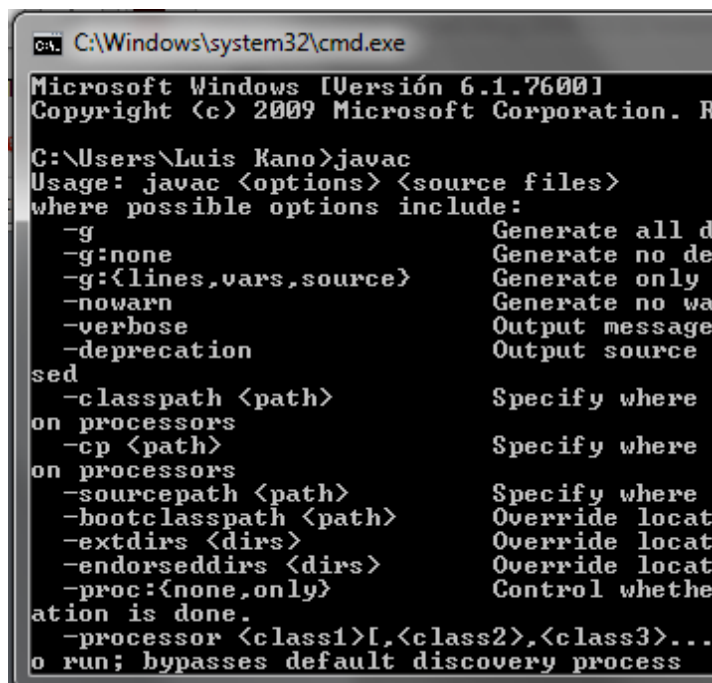


Buscamos en las Variables del sistema, la variable llamada Path y le damos clic en Editar



Se abrirá una nueva ventana y le agregamos esto: C:\Program Files\Java\jdk1.6.0\_18\bin si es que dejamos la ruta por default del instalador, si lo metiste en otra carpeta, deberás poner la dirección donde se encuentra instalado el JDK. Y damos Clic en Aceptar.

Ahora nos iremos a la Consola de Comandos de Windows 7, o al famoso CMD, una vez abierto escribimos javac y deberá mostrar este contenido.



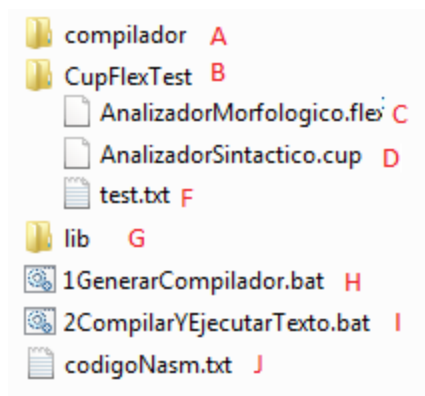
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. R

G:\Users\Luis Kano>javac
Usage: javac <options> <source files>
where possible options include:
-g Generate all de
-g:none Generate no de
-g:<lines,vars,source> Generate only s
-nowarn Generate no wa
-verbose Output message
-deprecation Output source
sed
-classpath <path> Specify where
on processors
-cp <path> Specify where
on processors
-sourcepath <path> Specify where
-bootclasspath <path> Override locat
-extdirs <dirs> Override locat
-endorseddirs <dirs> Override locat
-proc:<none,only> Control whethe
ation is done.
-processor <class1>[,<class2>,<class3>...
o run; bypasses default discovery process
```

Y si se muestra esto, quedó instalado el Java JDK en nuestro sistema de Windows 7, y podemos ejecutar nuestro compilador sin problemas.

## 2.2 Ejecución del compilador

Antes de nada, vamos a proceder a explicar el contenido de nuestro proyecto para así poder explicar más fácilmente como se ejecuta nuestro compilador. Para explicarlo, nos guiaremos con la siguiente imagen:



A) Contiene principalmente clases Java necesarias para la ejecución de nuestro compilador.

B) Contiene los archivos principales de nuestro sistema:

C) Es el analizador morfológico de nuestro compilador, se puede diferenciar porque es el .flex.

D) Es el analizador sintáctico de nuestro compilador nasm, lleva extensión .cup.

F) Es el código a compilar, aquí debemos escribir el código en lenguaje de alto nivel para que el compilador lo traduzca en código nasm.

G) Son las librerías necesarias para la compilación de nuestro compilador.

H) Es un script de windows genera el compilador que hemos diseñado. Habrá que darle si se ha realizado algún cambio en el código del compilador.

I) Es otro script de windows que lanza nuestro compilador ya generado, utilizará el código del punto “F” para depositar el resultado en el fichero del punto siguiente. Habrá que ejecutarlo si queremos compilar con nuestro compilador

J) Es el código resultante de la compilación realizada del fichero “F”. Se genera cuando ejecutamos el script del punto “I”

## 2.3 Fichero de prueba

Para realizar las pruebas del compilador, hemos utilizado el siguiente código:

— Test.txt

```
program {
    int uno;
    int dos,res;
    uno = 1;
    dos = 2;
    res = uno + dos;
    res = dos - uno;
    res = uno * dos;
    res = dos / uno;

    if(res != 3){
        res = res + 1;
    }else{
        res= res + 3;
    }
}
```

Y como resultado después de la ejecución hemos obtenido lo siguiente:

— **CodigoNasm.txt**

```
_uno resd 1
_dos resd 1
_res resd 1
__aux resd 1
segment .text
global _main
extern _print_int, _print_endofline, _scan_int

push dword 1

pop dword eax
mov dword [_uno],eax

push dword 2

pop dword eax
mov dword [_dos],eax

push dword _uno
pop dword eax
mov dword eax,[eax]
push dword eax

push dword _dos
pop dword eax
mov dword eax,[eax]
push dword eax

pop dword eax
pop dword ebx
add dword eax,ebx
push dword eax

pop dword eax
mov dword [_res],eax

push dword _dos
pop dword eax
mov dword eax,[eax]
push dword eax

push dword _uno
pop dword eax
mov dword eax,[eax]
push dword eax
```

```
pop dword ebx
pop dword eax
sub dword eax,ebx
push dword eax

pop dword eax
mov dword [_res],eax

push dword _uno
pop dword eax
mov dword eax,[eax]
push dword eax

push dword _dos
pop dword eax
mov dword eax,[eax]
push dword eax

pop dword eax
pop dword ebx
imul dword eax,ebx
push dword eax

pop dword eax
mov dword [_res],eax

push dword _dos
pop dword eax
mov dword eax,[eax]
push dword eax

push dword _uno
pop dword eax
mov dword eax,[eax]
push dword eax

pop dword ebx
pop dword eax
cdq
idiv dword ebx
push dword eax

pop dword eax
mov dword [_res],eax

push dword _res
pop dword eax
mov dword eax,[eax]
push dword eax

push dword 3
```

```
pop dword eax
pop dword ebx
cmp eax,ebx
jne near distinto_1
push dword 0
jmp fin_distinto_1
distinto_1: push dword 1
    fin_distinto_1:

condicional_1: pop dword eax
mov dword ebx,0
cmp eax,ebx
je near ppio_else_1

push dword _res
pop dword eax
mov dword eax,[eax]
push dword eax

push dword 1

pop dword eax
pop dword ebx
add dword eax,ebx
push dword eax

pop dword eax
mov dword [_res],eax

jmp near fin_condicional_1
ppio_else_1 :

push dword _res
pop dword eax
mov dword eax,[eax]
push dword eax

push dword 3

pop dword eax
pop dword ebx
add dword eax,ebx
push dword eax

pop dword eax
mov dword [_res],eax

fin_condicional_1:

ret
```

### 3. Explicación de los ficheros del compilador

En este apartado vamos a explicar los componentes programados más relevantes del compilador, presentando el código correspondiente, que irá acompañado por comentarios escritos en este formato.

#### 3.1 Analizador Morfológico

A continuación presentamos el fichero de Flex correspondiente con el analizador morfológico:

— AnalizadorMorfológico.flex

##### **CÓDIGO DE USUARIO:**

*En esta sección se declara el código que queremos copiar de forma literal al código generado (aparecerá en la parte superior del código generado por el lexer). Es un sitio óptimo para incluir comentarios javadoc (si no está presente se generará uno de forma automática y genérica), imports y declaraciones de package.*

*En este caso pertenecerá al package 'compilador', e importa las librerías de CUP.*

```
package compilador;
import java_cup.runtime.*;
```

```
%%
```

##### **OPCIONES Y DECLARACIONES (MACROS):**

##### **Opciones:**

- La clase generada se llamará *AnalizadorMorfológico*
- Emplea formato de codificación: *unicode*
- Activa el conteo de líneas, accesible en la variable *'yyline'*
- Activa el conteo de columnas, accesible en la variable *'yycolumn'*

```
%class AnalizadorMorfológico
%unicode
%line
%column
%cup
```

*Este bloque entre %{ y }% define código que será copiado de forma literal, al igual que el escrito en la sección de Código de usuario, se emplea para inicializar las variables o funciones auxiliares que necesitamos.*

*En concreto, estamos definiendo la función *symbol* que devolverá el *Symbol* correspondiente al tipo dado (el resto de parámetros: *linea*, *columna* y *token*) los obtendrá de las variables activadas arriba.*

```
%{
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn, yytext());
    }

    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
}
```

```
    }
}%}
```

**DECLARACIÓN DE MACROS:**

*En esta sección especifica las macros empleadas. Las macros son abreviaciones de expresiones regulares, que facilitan la creación de otras expresiones mas complejas, aunque no se pueden emplear como no terminales (recursivos o mutuamente recursivos), en cuyo caso se devolvería una excepción generando el JFlex*

```
letra = [a-zA-Z]
entero = [0-9]+
real = {entero}"."{entero}
realSinEntero = "."{entero}
realSinDecimal = {entero}"."
finDeLinea = \r|\n|\r\n
comentario = \/\/[^\n\r]*{finDeLinea}
idVariable = {letra}[a-zA-Z0-9]{0,29}
idVariableLarga = {letra}[a-zA-Z0-9]+
idVariableEntera = {entero}[a-zA-Z0-9]+
```

*De forma opcional aquí se puede declarar un estado léxico, que podrá ser empleado en las reglas léxicas.*

```
%%
```

**REGLAS LÉXICAS:**

*Esta es la sección de las reglas léxicas, contiene expresiones regulares y acciones que son ejecutadas cuando el scanner encuentra una coincidencia asociada a su expresión regular. Siempre ejecutará las acciones de la expresión regular coincidente de mayor longitud, y si la palabra cumple con dos expresiones regulares de igual longitud, se ejecutan las acciones de la primera expresión regular.*

*Todas las expresiones regulares devolverán un objeto de la clase Symbol (construido con el constructor que definimos arriba) correspondiente con el tipo que representan. Este tipo se codifica como un entero que sacamos accediendo a los atributos del objeto 'sym'.*

```
"program"      {return symbol(sym.PROGRAM);}
"if"           {return symbol(sym.IF);}
"else"         {return symbol(sym.ELSE);}
"while"        {return symbol(sym.WHILE);}
"for"          {return symbol(sym.FOR);}

"int"          {return symbol(sym.ENTERO);}
"boolean"      {return symbol(sym.BOOLEAN);}
"float"        {return symbol(sym.FLOAT);}

"true"         {return symbol(sym.CTE_BOOLEANA);}
"false"        {return symbol(sym.CTE_BOOLEANA);}

"scanf"        {return symbol(sym.SCANF);}
"cprintf"      {return symbol(sym.CPRINTF);}
"printf"       {return symbol(sym.PRINTF);}

"array"        {return symbol(sym.ARRAY);}
"function"     {return symbol(sym.FUNCTION);}
"return"       {return symbol(sym.RETURN);}

";"           {return symbol(sym.SEMI);}
","           {return symbol(sym.COMA);}
":"           {return symbol(sym.PUNTOS);}
```



```

"+"      {return symbol(sym.MAS);}
"- "     {return symbol(sym.MENOS);}
"/"      {return symbol(sym.ENTRE);}
"*"      {return symbol(sym.POR);}

"&&"     {return symbol(sym.AND);}
"|"      {return symbol(sym.OR);}

"!"      {return symbol(sym.NOT);}

"{"      {return symbol(sym.LEFT_LLAVE);}
"}"      {return symbol(sym.RIGHT_LLAVE);}
"("      {return symbol(sym.LEFT_PARENTESIS);}
")"      {return symbol(sym.RIGHT_PARENTESIS);}
"["      {return symbol(sym.LEFT_CORCHETE);}
"]"      {return symbol(sym.RIGHT_CORCHETE);}

"="      {return symbol(sym.ASIGNACION);}
"=="     {return symbol(sym.EQUALS);}
"!="     {return symbol(sym.NOT_EQUALS);}
">="     {return symbol(sym.GREATER_EQUALS);}
"<="     {return symbol(sym.LESS_EQUALS);}
"<"      {return symbol(sym.LESS);}
">"      {return symbol(sym.GREATER);}

```

**Los comentarios, tabulaciones, strings y fin de línea nos los saltamos (no aplicamos ninguna regla)**

```

{comentario}      {}
" "               {}
{finDeLinea}      {}
"\t"              {}

{entero}           {return symbol(sym.INT, new Integer(yytext()));}

{real}             {return symbol(sym.REAL, new Float(yytext()));}

{realSinEntero}    {return symbol(sym.REAL_SIN_INT, new Float(yytext()));}

{realSinDecimal}   {return symbol(sym.REAL_SIN_DEC, new Float(yytext()));}

{idVariable}       {return symbol(sym.ID_VAR);}

```

**En caso de no encontrar coincidencia reconociendo la entrada parseada con las expresiones regulares definidas arriba, se imprime por consola un error (con la fila y columna donde se ha producido) y se parará el parseo.**

```

.      {System.out.print("  Error  (fila  "+(yyline+1)+",  columna
"+(yycolumn+1)+"): simbolo "+yytext()+" no valido.");
      System.exit(0);}

```

## 3.1 Analizador Sintáctico

A continuación presentamos el fichero de Flex correspondiente con el analizador sintáctico:

— AnalizadorSintactico.cup

**Primero ponemos en lenguaje Java el package al que pertenece e importamos las librerías necesarias.**

```
package compilador;

import java_cup.runtime.*;
import java.io.*;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Stack;
import compilador.SimboloSemantico.Tipo;
```

**El código dentro de “action code” es código local al CUP, empleado en la gramática. Este código se emplea para generar el parser.java, pero no se copiará de forma literal. En concreto, estamos instanciando una tabla de símbolos que contiene como clave los tokens y los asocia a un SimboloSemantico (clase definida en el proyecto) Así mismo creamos los contadores necesarios para compilar (contadorIgual empleado en la comparacion EQUALS, y contadorDistinto para la NOTEQUALS), una pila para los ifs anidados y su contador correspondiente**

```
action code{
    HashMap<String,SimboloSemantico> tablaSimbolos = new
        HashMap<String,SimboloSemantico>();
    int contadorIgual = 1;
    int contadorDistinto = 1;
    Stack<Integer> pilaIf = new Stack<Integer>();
    int contadorIf = 1;

:};
```

**A diferencia del “action code” anterior, en el “parser code” todo lo que pongamos se copiará de forma literal en el parser.java generado. Es por tanto un sitio perfecto para definir funciones auxiliares (en nuestro caso las de reporte de error) y el ‘main’ del programa.**

**parser code {:**

**En el main preparamos la ejecución del compilador. En concreto, recibirá el texto de entrada como argumento en la ejecución.**

```
public static void main (String[] args) throws Exception {
    if (args.length == 0)
        new parser(new AnalizadorMorfologico(System.in)).parse();
    else {
        InputStream inputStream = new ByteArrayInputStream(args[0].getBytes());
        new parser(new AnalizadorMorfologico(inputStream)).parse();
    }
}
```

```
public void report_error(String message, Object info) {
```

```
    StringBuffer m = new StringBuffer("ERROR:");
```

```

        if (info instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
            if (s.left >= 0) {
                m.append(" se encontro un fallo en la linea "+(s.left+1));
                if (s.right >= 0)
                    m.append(" y columna "+(s.right+1));
            }
            if (s.sym >= 0)
                m.append(" por el simbolo \""+(getSimbolo(info)) + "\" : la sintaxis no
es correcta");
        }
    }
    System.out.println(m);
    System.exit(0);
}

```

**NOTA:** Esa funcion “*getSimbolo*” la empleamos para establecer una relación entre el número asignado a los símbolos terminales, en el fichero sym.java (generado por el CUP) y un string que representa lo que són realmente. Por tanto, dado el int de ese tipo, se devolverá un String con el nombre del terminal correspondiente. Esta función la empleamos en la generación de errores.

```

public String getSimbolo (Object info){

    java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
    String simbolo="UNDEFINED";

    switch(s.sym){

        case 0: simbolo= "EOF";
            break;

        case 1: simbolo= "ERROR";
            break;

        case 2: simbolo= "main";
            break;

        case 3: simbolo= "if";
            break;

        case 4: simbolo= "else";
            break;

        case 5: simbolo= "while";
            break;

        case 6: simbolo= "CTE_BOOLEANA";
            break;

        case 7: simbolo= "scanf";
            break;

        case 8: simbolo= "printf";
            break;

        case 9: simbolo= ";";
            break;

        case 10: simbolo= ",";
            break;
    }
}

```

```
case 11: simbolo= ":";
        break;

case 12: simbolo= "+";
        break;

case 13: simbolo= "-";
        break;

case 14: simbolo= "/";
        break;

case 15: simbolo= "*";
        break;

case 16: simbolo= "&&";
        break;

case 17: simbolo= "||";
        break;

case 18: simbolo= "!";
        break;

case 19: simbolo= "{";
        break;

case 20: simbolo= "}";
        break;

case 21: simbolo= "(";
        break;

case 22: simbolo= ")";
        break;

case 23: simbolo= "=";
        break;

case 24: simbolo= "==" ;
        break;

case 25: simbolo= "!=";
        break;

case 26: simbolo= ">=";
        break;

case 27: simbolo= "<=";
        break;

case 28: simbolo= "<";
        break;

case 29: simbolo= ">";
        break;

case 30: simbolo= "ID_VAR";
        break;
```

```

        case 31: simbolo= "cprintf";
            break;

        case 32: simbolo= "function";
            break;

        case 33: simbolo= "array";
            break;

        case 34: simbolo= "[";
            break;

        case 35: simbolo= "]";
            break;

        case 36: simbolo= "return";
            break;

        case 37: simbolo="ENTERO";
            break;

        case 38: simbolo="BOOLEAN";
            break;

        case 39: simbolo="FLOAT";
            break;

        case 41: simbolo="INT";
            break;

        case 42: simbolo="REAL";
            break;

        case 43: simbolo="REAL_SIN_INT";
            break;

        case 44: simbolo="REAL_SIN_DEC";
            break;

    }

    return simbolo;
}

public void report_fatal_error(String message, Object info) {
    System.out.println(message);
}

:};

```

**Declaración de los símbolos terminales y no terminales de la gramática. Nótese que los símbolos terminales han de coincidir con los devueltos que instanciamos en el fichero flex.**

```

terminal String      PROGRAM, IF, ELSE, WHILE, CTE_BOOLEANA, CONST,
                     SCANF, PRINTF, SEMI, COMA, PUNTOS, MAS, MENOS, ENTRE, POR,
                     AND, OR, NOT, LEFT_LLAVE, RIGHT_LLAVE,
                     LEFT_PARENTESIS, RIGHT_PARENTESIS, ASIGNACION,
                     EQUALS, NOT_EQUALS, GREATER_EQUALS, LESS_EQUALS, LESS, MALLOC,
FREE, SIZE, CONTAINS,

```

```

        GREATER, ID_VAR, CPRINTF, FUNCTION, ARRAY, LEFT_CORCHETE,
        RIGHT_CORCHETE, RETURN, ENTERO, BOOLEAN, FLOAT, FOR;

terminal Integer      INT;
terminal Float        REAL, REAL_SIN_INT, REAL_SIN_DEC;

non terminal String    programa, declaraciones, sentencias, declaracion,
                        identificadores, sentencia, sentencia_simple,
                        estructura_control, asignacion, lectura, escritura,
                        condicional, condicionalParcial, bucle, comparacion,
                        cte_real, funciones, funcion, declaraciones_funcion,
                        declaracion_funcion, retorno;
non terminal Tipo      clase, exp, constante, elemento_vector, tipo_coleccion, tipo,
                        parametro;
non terminal ArrayList<Tipo>  parametros, resto_parametros, lista_expresiones,
                        resto_lista_expresiones;

```

**En este apartado establecemos las precedencias entre las operaciones aritméticas y los limitadores de bloque (paréntesis, llaves...)**

```

precedence left OR;
precedence left AND;
precedence left EQUALS, NOT_EQUALS, GREATER_EQUALS, LESS_EQUALS, LESS, GREATER;
precedence left ASIGNACION;
precedence left MAS, MENOS;
precedence left POR, ENTRE;
precedence left COMA;
precedence left NOT;
precedence left LEFT_PARENTESIS, RIGHT_PARENTESIS, LEFT_LLAVE, RIGHT_LLAVE,
LEFT_CORCHETE, RIGHT_CORCHETE;

```

#### **GRAMÁTICA DEL LENGUAJE:**

**Aquí se declara la gramática asociada al lenguaje, con sus reglas correspondientes,**

**donde vamos a imprimir información (tanto de la reducción como de los errores provocados), y donde se generará directamente el código NASM compilado. Este código se escribirá en el fichero designado por medio de la clase `EscritorFichero` y su método `escribir (String textoAEscribir)`.**

**La gramática se ha copiado tal como dictaba el enunciado, pero solo se han implementado las acciones correspondientes a las operaciones aritméticas de los parámetros de tipo entero (INT) y los condicionales (IF y ELSE), junto a las reglas mínimas necesarias para compilar un programa funcional completo.**

```

programa      ::= PROGRAM:m LEFT_LLAVE:ll declaraciones:d funciones:f sentencias:s
RIGHT_LLAVE:r1
                {:      EscritorFichero.escribir("ret");
                    RESULT = m + ll + d + f + s + r1;
                    System.out.println("Reduce a programa");

                };

```

**Aquí se importan las librerías necesarias para la ejecución.**

```

declaraciones ::= declaracion:d
                {:      EscritorFichero.escribir("__aux resd 1 \n__auxElemVector
resd 1 \nsegment .text \nglobal _main "+
                    "\nextern _print_int, _print_endofline,
_scan_int\n");

                    RESULT = d;
                    System.out.println("Reduce a declaraciones por
declaracion");
                :}

```

```

        | declaracion:d declaraciones:dd
        {:      RESULT = d + dd;
          System.out.println("Reduce a declaraciones por declaracion
declaraciones");      :}};

declaracion ::= clase:t identificadores:n SEMI:s
        {:      RESULT = t + " " + n + s;
          String ids[] = n.split(",");

          for(String id: ids){
            if(tablaSimbolos.containsKey(id)){

              System.out.println("Error (fila
" +(nleft+1)+",columna " +(nright+1)+"): la variable " + id + " ya ha sido declarada");
              System.exit(0);
            }
            else{
              if(t == Tipo.INT ){
                tablaSimbolos.put(id,new
SimboloSemantico(id,t));
EscriptorFichero.escribir("_"+id+" resd 1\n");
              }
            }
          }
          System.out.println("Reduce a declaracion");      :}};

clase ::= tipo:t
        {:      RESULT = t;
          System.out.println("Reduce a clase por tipo");      :}
        | tipo_coleccion:t
        {:      RESULT = t;
          System.out.println("Reduce a clase por tipo_coleccion");
        :}};

tipo ::= ENTERO
        {:      RESULT = Tipo.INT;
        :}
        | BOOLEAN
        | FLOAT
        ;

tipo_coleccion ::= ARRAY:a tipo:t LEFT_CORCHETE:lc INT:i RIGHT_CORCHETE:rc
        ;

identificadores ::= ID_VAR:i
        {:      RESULT = i;
          System.out.println("Reduce a identificadores por ID_VAR");
        :}
        | ID_VAR:i COMA:c identificadores:id
        {:      RESULT = i + c + id;
        :}};

funciones ::= funcion:f funciones:ff| ;

funcion ::= FUNCTION:f clase:c ID_VAR:i LEFT_PARENTESES:lp ;

parametros ::= parametro:p resto_parametros:r
        |

```

```

;
resto_parametros ::= SEMI:s parametro:p resto_parametros:r
|
;
parametro ::= tipo:t ID_VAR:i
;
declaraciones_funcion ::= declaracion_funcion:d declaraciones_funcion:dd
|
;
declaracion_funcion ::= tipo:tu ID_VAR:i SEMI:s
;
sentencias ::= sentencia:s
{
    RESULT = s;
    System.out.println("Reduce a sentencias por sentencia");
:}
| sentencia:s sentencias:ss
{
    RESULT = s + ss;
    System.out.println("Reduce a sentencias por sentencia
sentencias"); :};

sentencia ::= sentencia_simple:ss SEMI:s
{
    RESULT = ss + s;
    System.out.println("Reduce a sentencia por sentencia_simple
SEMI"); :}
| estructura_control:ec
{
    RESULT = ec;
    System.out.println("Reduce a sentencia por
estructura_control"); :};

sentencia_simple ::= asignacion:a
{
    RESULT = a;
    System.out.println("Reduce a sentencia_simple por
asignacion"); :}
| lectura:l
| escritura:e
| retorno:r
;

asignacion ::= ID_VAR:i ASIGNACION:a exp:e
{
    RESULT = i + a + e;
    Tipo tipo = null;

    if(tablaSimbolos.containsKey(i)){
        tipo = tablaSimbolos.get(i).getTipo();
        EscritorFichero.escribir("\npop dword eax\nmov
dword [_"+i+"],eax\n");
    }else{
        System.out.println("Error (fila
"+(ileft+1)+",columna "+(iright+1)+"): la variable: "+ i + " no ha sido declarada");
        System.exit(0);
    }

    if(tipo != e){

```



```

                                System.out.println("Error (fila
"+(aleft+1)+",columna "+(aright+1)+"): los tipos de ambos lados de la asignacion no
corresponden: a izquierda: "
                                + tipo + " a derecha: " + e);
                                System.exit(0);
                                }

                                System.out.println("Reduce a asignacion por "+i+ a +
"exp");      :}
                                | elemento_vector:ev ASIGNACION:a exp:e
                                {:      if(ev != e){
                                System.out.println("Error (fila
"+(aleft+1)+",columna "+(aright+1)+"): los tipos de ambos lados de la asignacion no
corresponden: a izquierda: "
                                + ev + " a derecha: " + e);
                                System.exit(0);
                                }
                                EscritorFichero.escribir("\npop dword eax \npop dword
ebx \nmov dword ebx,[_auxElemVector] \nmov dword [ebx],eax");
                                RESULT = ev + a + e;

                                System.out.println("Reduce a asignacion por elemento_vector
ASIGNACION exp");      :};

elemento_vector      ::= ID_VAR:i LEFT_CORCHETE:lc exp:e RIGHT_CORCHETE:rc
;

estructura_control   ::= condicional:c
{:      RESULT = c;
System.out.println("Reduce a estructura_control por
condicional");      :}
| bucle:b
{:      RESULT = b;
System.out.println("Reduce a estructura_control por
bucle");      :};

condicional           ::= condicionalParcial:cp
{: Integer cuenta = pilaIf.pop();
EscritorFichero.escribir("fin_condicional_"+cuenta+":
\nppio_else_"+cuenta+": \n");
RESULT = cp; :}
| condicionalParcial:cp ELSE:e
{: Integer cuenta = pilaIf.pop();
EscritorFichero.escribir("ppio_else_"+cuenta+": ");
pilaIf.push(cuenta);
:} LEFT_LLAVE:l12 sentencias:s2 RIGHT_LLAVE:r12
{:
EscritorFichero.escribir("fin_condicional_"+pilaIf.pop()+":
\n");
RESULT = cp + e + l12 + s2 + r12;
:};

condicionalParcial::= IF:i LEFT_PARENTEESIS:lp exp:ex RIGHT_PARENTEESIS:rp {:
EscritorFichero.escribir("condicional_"+contadorIf+": pop dword
eax \nmov dword ebx,0 \ncmp eax,ebx "
                                +"\nje near ppio_else_"+contadorIf+ "\n");
pilaIf.push(new Integer(contadorIf));
contadorIf++;
:} LEFT_LLAVE:l11 sentencias:s1 RIGHT_LLAVE:r11
{:      if(ex != Tipo.BOOLEAN){
                                System.out.println("Error (fila

```

```

" +(exleft+1)+",columna " +(exright+1)+"): la expresion del condicional es incorrecta. Se
espera: BOOLEAN, se encuentra: "+ex);
        System.exit(0);
    }
    Integer cuenta = pilaIf.pop();
    EscritorFichero.escribir("jmp near
fin_condicional_"+cuenta);
    pilaIf.push(cuenta);
    RESULT = i + lp + ex + rp + ll1 + s1 + r11;
    System.out.println("Reduce a condicional por IF ELSE");
};

bucle ::= WHILE:w
        LEFT_PARENTESIS:lp exp:e RIGHT_PARENTESIS:r

        LEFT_LLAVE:ll sentencias:s RIGHT_LLAVE:r1

        | FOR:f LEFT_PARENTESIS:lp ID_VAR:i PUNTOS:p ID_VAR:a
RIGHT_PARENTESIS:rp

        LEFT_LLAVE:ll sentencias:s RIGHT_LLAVE:r1 ;

lectura ::= SCANF:s ID_VAR:i
;

escritura ::= CPRINTF:c ID_VAR:i

        |CPRINTF:c elemento_vector:ev

        | PRINTF:p exp:e
;

retorno ::= RETURN:r exp:e
;

constante ::= CTE_BOOLEANA:b
        | INT:i
        {:      EscritorFichero.escribir("push dword "+i+"\n");
                RESULT = Tipo.INT;
                System.out.println("Reduce a constante por INT");      :}
        | cte_real:r
;

cte_real ::= REAL:r

        | REAL_SIN_INT:r

        | REAL_SIN_DEC:r
;

exp ::= exp:e1 MAS:m exp:e2
        {:      if(e1 == Tipo.INT && e2 == Tipo.INT){
                EscritorFichero.escribir("pop dword eax \npop dword
ebx \nadd dword eax,ebx \npush dword eax\n");
                RESULT = Tipo.INT;
            }else{
                System.out.println("Error (fila
" +(mleft+1)+",columna " +(mright+1)+"): los tipos de la expresion suma no son
validos.");
                System.exit(0);
            }
        }
    
```

```

        }
        System.out.println("Reduce a exp por exp MAS exp");    :}
    | exp:e1 MENOS:m exp:e2
        {:      if(e1 == Tipo.INT && e2 == Tipo.INT){
                EscritorFichero.escribir("pop dword ebx \npop dword
eax \nsub dword eax,ebx \npush dword eax\n");
                RESULT = Tipo.INT;
            }else{
                System.out.println("Error (fila
"+"(mleft+1)+",columna "+(mright+1)+"): los tipos de la expresion resta no son
validos");

                System.exit(0);
            }
        }
        System.out.println("Reduce a exp por exp MENOS exp"); :}
    | exp:e1 ENTRE:e exp:e2
        {:      if(e1 == Tipo.INT && e2 == Tipo.INT){
                EscritorFichero.escribir("pop dword ebx \npop
dword eax \ncdq \nidiv dword ebx \npush dword eax\n");
                RESULT = Tipo.INT;
            } else{
                System.out.println("Error (fila
"+"(eleft+1)+",columna "+(eright+1)+"): los tipos de la expresion division no son
validos");

                System.exit(0);
            }
        }
        System.out.println("Reduce a exp por exp ENTRE exp"); :}
    | exp:e1 POR:p exp:e2
        {:      if(e1 == Tipo.INT && e2 == Tipo.INT){
                EscritorFichero.escribir("pop dword eax \npop
dword ebx \nimul dword eax,ebx \npush dword eax\n");
                RESULT = Tipo.INT;
            } else{
                System.out.println("Error (fila
"+"(pleft+1)+",columna "+(pright+1)+"): los tipos de la expresion multiplicacion no son
validos");

                System.exit(0);
            }
        }
        System.out.println("Reduce a exp por exp POR exp");    :}
    | MENOS:m exp:e

    | exp:e1 AND:a exp:e2

    | exp:e1 OR:o exp:e2

    | NOT:n exp:e

    | ID_VAR:i
        {:      if(tablaSimbolos.containsKey(i)) {
                RESULT = tablaSimbolos.get(i).getTipo();
                EscritorFichero.escribir("\npush dword
_ "+"i+" \npop dword eax \nmov dword eax,[eax] \npush dword eax\n");
            } else {
                System.out.println("Error (fila
"+"(ileft+1)+",columna "+(iright+1)+"): la variable '" + i + "' no ha sido declarada");
                System.exit(0);
            }
        }
        System.out.println("Reduce a exp por ID_VAR");    :}
    | constante:c
        {:
            RESULT = c;

```

```

        System.out.println("Reduce a exp por constante");      :}

    | LEFT_PARENTESES:lp exp:e RIGHT_PARENTESES:rp
      {:
        RESULT = e;
        System.out.println("Reduce a exp por LEFT_PARENTESES exp
RIGHT_PARENTESES"); :}
    | comparacion:c
      {:
        RESULT = Tipo.BOOLEAN;
        System.out.println("Reduce a exp por comparacion");    :}
    | elemento_vector:ev

    | ID_VAR:i LEFT_PARENTESES:lp lista_expresiones:le RIGHT_PARENTESES:rp
      ;

lista_expresiones ::= exp:e resto_lista_expresiones:rle
    |
    ;

resto_lista_expresiones ::= COMA:c exp:e resto_lista_expresiones:rle
    |
    ;

comparacion ::= exp:e1 EQUALS:eq exp:e2
    {:
        if(e1!=e2){
            System.out.println("Error (fila
"+(eqleft+1)+",columna "+(eqright+1)+"): los tipos de ambos lados de la comparacion no
corresponden: a izquierda: "
                                + e1 + " a derecha: " + e2);
            System.exit(0);
        }
        EscritorFichero.escribir("pop dword eax \npop dword ebx
\n cmp eax,ebx "
                                + "\nje near igual_"+contadorIgual+" \npush
dword 0 \n jmp fin_igual_"+contadorIgual
                                + "\n igual_"+contadorIgual+": push dword 1
\n fin_igual_"+contadorIgual+": \n");

        contadorIgual++;
        RESULT = e1 + eq + e2;
        System.out.println("Reduce a exp por exp EQUALS exp");
    :}

    | exp:e1 NOT_EQUALS:neq exp:e2
      {:
        if(e1!=e2){
            System.out.println("Error (fila
"+(neqleft+1)+",columna "+(neqright+1)+"): los tipos de ambos lados de la comparacion
no corresponden: a izquierda: "
                                + e1 + " a derecha: " + e2);
            System.exit(0);
        }

        EscritorFichero.escribir("pop dword eax \npop dword
ebx \n cmp eax,ebx "
                                + "\njne near distinto_"+contadorDistinto+"
\n push dword 0 \n jmp fin_distinto_"+contadorDistinto
                                + "\n distinto_"+contadorDistinto+": push
dword 1 \n fin_distinto_"+contadorDistinto+": \n");

```

```
                                contadorDistinto++;  
                                RESULT = e1 + neq + e2;  
                                System.out.println("Reduce a exp por exp NOT_EQUALS  
exp");:;}  
                                | exp:e1 LESS_EQUALS:le exp:e2  
                                | exp:e1 GREATER_EQUALS:ge exp:e2  
                                | exp:e1 LESS:l exp:e2  
                                | exp:e1 GREATER:g exp:e2  
                                ;
```