Información de elementos léxicos:

Token	Descripción	Expresión regular	Ejemplo
Bucle	Reconoce las directivas de cada bucle	while | for | do	while
Numero	Reconoce números enteros sin signos	[0-9]+	45
Variable	Reconoce cadenas de caracteres como nombres de variables	[A-Za-z][a-zA-Z0-9_]*	Ac_D9
True	Reconoce la directiva true	true	true
False	Reconoce la directiva false	false	false
Entero	Reconoce la directiva int	int	int
Boolean	Reconoce la directiva boolean	boolean	boolean
NoRetorno	Reconoce la directiva void	void	void
Retorno	Reconoce la directiva return	return	return
Aritmetica	Reconoce los símbolos aritméticos	+|*|-|/	+
Relacional	Reconoce los símbolos relacionales de las expresiones	<|<=|>|>=|==|!=	<
Logica	Reconoce los símbolos lógicos de las expresiones	&&| || |!	&&
Parentesis	Reconoce el paréntesis de apertura y cierre en una expresión	(|)	(
Llave	Reconoce la llave abierta y cerrada al comienzo y al final de un método o clase	{ | }	{
Puntuación	Reconoce símbolos de puntuación	, | ; | .	;
Clase	Reconoce la directiva class	class	class
Public	Reconoce la directiva public	public	public
Static	Reconoce la directiva static	static	static

Incremento	Reconoce una variable que se incrementa	Variable ++ | variable +=	Var++
Decremento	Reconoce una variable que se decrementa	Variable -- | variable -=	Var--
Metodo	Reconoce: -Si la palabra está precedida por "public static", reconoce la declaración de un método. - En caso contrario reconoce, la llamada a un método	Variable(	calculaValor(
Asignacion	Reconoce el símbolo de asignación	=	=
SaltoDeLinea	Reconoce los saltos de línea	\n|\r|\r\n	
Espacio	Reconoce los espacios en blanco	" "	
Main	Reconoce la cabecera de los métodos main	main(	main(

Manual de usuario:

Desde Makefile (reglas):

- compile_and_run. Genera el archivo .java, lo compila y lo ejecuta y lee el fichero llamado "prueba.txt". Para leer otro fichero, ejecutar make compile_and_run input=<archivo>.
- run. Ejecuta el archivo ya compilado y lee el fichero llamado "prueba.txt". Para leer otro fichero realizar los pasos mencionados anteriormente.
- clean. Elimina el archivo .java y las clases generadas tras la compilación.

Sin Makefile:

1. Crear el archivo .java utilizando jflex:
   > jflex AnalizadorLexico.flex
2. Comiplar el archivo .java generado:
   > javac AnalizadorLexico.java
3. Ejecutar la clase compilada introduciendo como argumento el fichero a leer deseado:
   > java AnalizadorLexico <archivo>

Código de Jflex:

```
%%
%public
%class AnalizadorLexico
%8bit
%standalone
%{

 /* Código personalizado */
 // Variable utilizada para guardar la/las última/últimas palabras
reconocidas y ayudar a reconocer expresiones
 public String ultimaPalabra = "";

 // Método para realizar una impresión rápida con el formato solicitado
 public void imprimir(String lexema, String token){
        System.out.println("Nombre léxico: " + token + " => cadena
reconocida: " + lexema + ".");
 }
%}

/* Inicio de Expresiones regulares */
 Numero = [0-9]+
 Entero = "int"
 Boolean = "boolean"
 Retorno = "return"
 NoRetorno = "void"
 Variable = [a-zA-Z] [a-zA-Z0-9_]*
 SaltoDeLinea = \n|\r|\r\n
 Asignacion = "="
 Puntuacion = ";" | "," |"."
 Bucle = "for" | "while" | "do"
 Aritmetica = "*"|"+"|"-"|"/"
 Relacional = "<"|"<="|">"|">="|"=="|"!="
 Logica = "&&"|"||"|"!"
 Espacio = " "
 Parentesis = "(" | ")"
 Llave = "{" | "}"
 Verdadero = "True"
 Falso = "False"
 Incremento = {Variable} "++" | {Variable} "+="
 Decremento = {Variable} "--" | {Variable} "-="
 Main = "main("
 Public = "public"
 Static = "static"
 Class = "class"
 Metodo = {Variable} "("
```

```
/* Finaliza expresiones regulares */

%%
/* Finaliza la sección de declaraciones de JFlex */

/* Inicia sección de reglas */

// Cada regla está formada por una {expresión} espacio {código}

{Numero} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "NÚMERO");
}

{SaltoDeLinea} {
 ultimaPalabra = yytext();
 imprimir("Enter", "NUEVA_LINEA");
}

{Asignacion} {
 ultimaPalabra = yytext();
 imprimir(yytext(),"ASIGNACIÓN");
}

{Puntuacion} {
 ultimaPalabra = yytext();
 imprimir(yytext(),"PUNTUACIÓN");
}

{Bucle} {
 ultimaPalabra = yytext();
 imprimir(yytext(),"BUCLE");
}

{Aritmetica} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "SÍMBOLO");
}

{Logica} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "LÓGICA");
}

{Relacional} {
```

```
 ultimaPalabra = yytext();
 imprimir(yytext(), "RELACIONAL");
}

{Espacio} {
 // Ignorar cuando se ingrese un espacio
}

{Entero} {
 if(ultimaPalabra.equals("public static")){
       ultimaPalabra = ultimaPalabra +" "+ yytext();
 }else{
       ultimaPalabra = yytext();
       imprimir(yytext(), "ENTERO");
 }
}

{Boolean} {
 if(ultimaPalabra.equals("public static")){
       ultimaPalabra = ultimaPalabra +" "+ yytext();
 }else{
       ultimaPalabra = yytext();
       imprimir(yytext(), "BOOLEAN");
 }
}

{Retorno} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "RETORNO");
}

{NoRetorno} {
 if(ultimaPalabra.equals("public static")){
       ultimaPalabra = ultimaPalabra +" "+ yytext();
 }else{
       ultimaPalabra = yytext();
       imprimir(yytext(), "VOID");
 }
}

{Class} {
 if(ultimaPalabra.equals("public")) ultimaPalabra = ultimaPalabra +" "+
yytext();
}

{Static} {
```

```
 if(ultimaPalabra.equals("public")) ultimaPalabra = ultimaPalabra +" "+
yytext();
}

{Public} {
 ultimaPalabra = yytext();
}

{Parentesis} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "PARENTESIS");
}

{Llave} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "LLAVE");
}

{Verdadero} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "TRUE");
}

{Falso} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "FALSE");
}

{Decremento} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "DECREMENTO");
}

{Incremento} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "INCREMENTO");
}

{Main} {
 ultimaPalabra = yytext();
 imprimir(yytext(), "MAIN");
}
{Metodo} {
 if(ultimaPalabra.equals("public static void") ||
ultimaPalabra.equals("public static int") || ultimaPalabra.equals("public
static boolean")){
```

```
        ultimaPalabra = yytext();
        imprimir(yytext(), "METODO");
 }else{
        ultimaPalabra = yytext();
        imprimir(yytext(), "LLAMADA A MÉTODO");
 }
}
{Variable} {
 if(ultimaPalabra.equals("public class")){
        ultimaPalabra = yytext();
        imprimir(yytext(), "CLASE");
 }else{
        ultimaPalabra = yytext();
        imprimir(yytext(), "VARIABLE");
 }
}
```

Output del programa:

CLASE => Ejem_TAC

LLAVE => {

NUEVA_LINEA => \n

METODO => comparar(

ENTERO => int

VARIABLE => valor

PUNTUACIÓN => ,

ENTERO => int

VARIABLE => tope

PARENTESIS =>)

LLAVE => {

NUEVA_LINEA => \n

ENTERO => int

VARIABLE => aux

PUNTUACIÓN => ;

NUEVA_LINEA => \n

VARIABLE => aux

ASIGNACIÓN => =

LLAMADA A MÉTODO => calculaValor(

VARIABLE => valor

PARENTESIS =>)

PUNTUACIÓN => ;

NUEVA_LINEA => \n

RETORNO => return

PARENTESIS => (

PARENTESIS => (

VARIABLE => tope

RELACIONAL => <

NÚMERO => 5

SÍMBOLO => *

NÚMERO => 2

PARENTESIS =>)

LÓGICA => ||

PARENTESIS => (

VARIABLE => aux

RELACIONAL => <

NÚMERO => 1

PARENTESIS =>)

PARENTESIS =>)

LÓGICA => &&

VARIABLE => true

PUNTUACIÓN => ;

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

METODO => calculaValor(

ENTERO => int

VARIABLE => valor1

PARENTESIS =>)

LLAVE => {

NUEVA_LINEA => \n

ENTERO => int

VARIABLE => aux

ASIGNACIÓN => =

NÚMERO => 1

PUNTUACIÓN => ;

NUEVA_LINEA => \n

BUCLE => for

PARENTESIS => (

ENTERO => int

VARIABLE => i

ASIGNACIÓN => =

NÚMERO => 0

PUNTUACIÓN => ;

VARIABLE => i

RELACIONAL => <

VARIABLE => valor1

PUNTUACIÓN => ;

INCREMENTO => i++

PARENTESIS =>)

LLAVE => {

NUEVA_LINEA => \n

BUCLE => for

PARENTESIS => (

ENTERO => int

VARIABLE => j

ASIGNACIÓN => =

VARIABLE => i

PUNTUACIÓN => ;

VARIABLE => j

RELACIONAL => <

VARIABLE => i

PUNTUACIÓN => ;

DECREMENTO => j--

PARENTESIS =>)

LLAVE => {

NUEVA_LINEA => \n

VARIABLE => aux

ASIGNACIÓN => =

VARIABLE => aux

SÍMBOLO => +

NÚMERO => 2

PUNTUACIÓN => ;

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

VARIABLE => aux

ASIGNACIÓN => =

VARIABLE => aux

SÍMBOLO => +

NÚMERO => 1

PUNTUACIÓN => ;

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

RETORNO => return

VARIABLE => aux

PUNTUACIÓN => ;

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

MAIN => main(

PARENTESIS =>)

LLAVE => {

NUEVA_LINEA => \n

BOOLEAN => boolean

VARIABLE => esCierto

PUNTUACIÓN => ;

NUEVA_LINEA => \n

VARIABLE => esCierto

ASIGNACIÓN => =

LLAMADA A MÉTODO => comparar(

NÚMERO => 2

PUNTUACIÓN => ,

NÚMERO => 5

PARENTESIS =>)

PUNTUACIÓN => ;

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

LLAVE => }

NUEVA_LINEA => \n

Explicación:

El programa reconoce una serie de expresiones léxicas que son palabras o secuencias de palabras (en los casos de las clases y declaraciones de métodos). Las explicaciones de cada expresión léxica se encuentran en la tabla del comienzo.