



SEP

SNEST

DGEST

INSTITUTO TECNOLÓGICO DE TOLUCA

Ingeniería en Sistemas Computacionales

Lenguajes y Autómatas 2

Diseño del lenguaje Coffe Code

Integrantes del equipo Ruby:

| | |
|-------------------------------|----------|
| Sánchez Valdin Kevin Ivan | 17280378 |
| Solís Robles Alondra Veronica | 17280333 |

Profesora:

Martha Martínez Moreno

Metepec, Estado de México, junio del 2021

Índice

| | |
|--|-----------|
| CAPITULO I. ANTECEDENTES DEL LENGUAJE | 6 |
| Nombre del lenguaje | 6 |
| Objetivo del lenguaje | 6 |
| Símbolos del lenguaje | 7 |
| CAPITULO II. ANÁLISIS LÉXICO | 9 |
| Expresiones regulares | 9 |
| Resultados del análisis léxico | 12 |
| CAPITULO III. ANÁLISIS SINTÁCTICO..... | 13 |
| Gramáticas descritas y ejemplos | 13 |
| Resultados del análisis sintáctico | 20 |
| CAPITULO IV. ANÁLISIS SEMÁNTICO | 21 |
| Validación semántica implementada | 21 |
| Métodos utilizados | 24 |
| Resultado de la validación semántica | 66 |
| CAPITULO V. ANALISIS INTERMEDIO | 69 |
| Ejemplos de resultados | 71 |
| Ejemplo 1..... | 71 |
| Ejemplo 2..... | 73 |
| Ejemplo 3..... | 76 |
| Código explicado | 78 |
| CAPITULO VI. CÓDIGO OPTIMIZADO | 86 |
| Ejemplo de optimización | 87 |

| | |
|---|----------------|
| Ejecución del código fuente..... | 88 |
| Resultado de la optimización – fuente | 89 |
| Ejecución del código optimizado | 90 |
| Resultados de la optimización – optimizado..... | 92 |
| Técnicas de Optimización | 93 |
| Código explicado | 94 |
| CAPITULO VII. INSTALACIÓN | 108 |
| CAPITULO VIII. MANUAL DE USUARIO..... | 114 |
| ANEXOS | 126 |
| Código ejemplo (Análisis léxico) | 126 |
| Código ejemplo (Análisis sintáctico) | 128 |
| Código ejemplo (Análisis semántico) | 130 |
| Código ejemplo (Generación del código intermedio) | 132 |
| FUENTES DE REFERENCIA | 135 |

Índice de figuras

| | |
|---|----|
| Figura 1 Código fuente del léxico | 12 |
| Figura 2 Resultado del análisis léxico | 12 |
| Figura 3 Código fuente del análisis sintáctico | 20 |
| Figura 4 Resultado del análisis sintáctico | 20 |
| Figura 5 Ejecución del código fuente para el análisis semántico con errores | 67 |
| Figura 6 Tabla de símbolos | 67 |
| Figura 7 Resultado del análisis semántico | 68 |
| Figura 8 Código fuente para el análisis intermedio | 69 |
| Figura 9 Resultado de la generación | 70 |
| Figura 10 Código fuente de la generación intermedio, ejemplo 1 | 72 |
| Figura 11 Ejemplo 1 | 73 |
| Figura 12 Código fuente de la generación intermedio, ejemplo 2 | 74 |
| Figura 13 Ejemplo 2 | 75 |
| Figura 14 Código fuente de la generación intermedio, ejemplo 3 | 76 |
| Figura 15 Ejemplo 3 | 77 |
| Figura 16 Impresión de la generación del código Intermedio | 78 |
| Figura 17 Llenar el ArrayList de tokens | 79 |
| Figura 18 Método conversionArrayCola() | 79 |
| Figura 19 Método recibeTokens() | 80 |
| Figura 20 Método posfijo(), parte 1 | 81 |
| Figura 21 Método posfijo(), parte 2 | 82 |
| Figura 22 Método insertaPOperadores() | 82 |
| Figura 23 Método prioridad() | 83 |
| Figura 24 Método isOperadorS() | 83 |
| Figura 25 Método operaciones () | 84 |
| Figura 26 Método operacionesIntermedio() | 85 |
| Figura 27 Método expresionFinal() | 85 |
| Figura 28 Pantalla de resultado del código optimizado | 86 |
| Figura 29 Código fuente | 87 |
| Figura 30 Archivos generados | 88 |
| Figura 31 Ejecución del código fuente | 88 |
| Figura 32 Resultado del código optimizado | 89 |
| Figura 33 Código optimizado | 90 |
| Figura 34 Ejecución del código optimizado | 91 |
| Figura 35 Archivos código optimizado | 91 |
| Figura 36 Resultados del código optimizado | 92 |
| Figura 37 Método optimizacion() | 94 |
| Figura 38 Método optimizaAsignacionesInnecesarias parte 1 | 95 |
| Figura 39 Método optimizaAsignacionesInnecesarias parte 2 | 95 |
| Figura 40 Método optimizaAsignacionesInnecesarias parte 3 | 96 |

| | |
|---|-----|
| Figura 41 Método optimizaCommetarios..... | 96 |
| Figura 42 imprimeOptimizacion..... | 97 |
| Figura 43 Método optimizaOperaciones() | 97 |
| Figura 44 Método optimizaVariableOp() parte 1..... | 98 |
| Figura 45 Método optimizaVariableOp() parte 2..... | 98 |
| Figura 46 Método optimizaVariablesOp() parte 3..... | 99 |
| Figura 47 Método optimizaVariablesOp() parte 4..... | 100 |
| Figura 48 Método optimizaEstructura() parte 1 | 101 |
| Figura 49 Método optimizaEstructura() parte 2 | 102 |
| Figura 50 Método optimizaVariablesSinUso() parte 1 | 103 |
| Figura 51 Método optimizaVariablesSinUso() parte 2 | 103 |
| Figura 52 Método optimizaVariablesSinUso() parte 3 | 104 |
| Figura 53 Método optimizaVariablesSinUso() parte 4 | 105 |
| Figura 54 Método optimizaVariablesSinUso() parte 5 | 106 |
| Figura 55 Método optimizaVariablesSinUso() parte 6 | 106 |
| Figura 56 Método optimizaVariablesSinUso() parte 7 | 107 |
| Figura 57 Método optimizaVariablesSinUso() parte 8 | 107 |
| Figura 58 Ejecutar como administrador..... | 108 |
| Figura 59 Seleccionar idioma de instalación | 108 |
| Figura 60 Seleccionar directorio destino de instalación | 109 |
| Figura 61 Crear un icono en el escritorio | 109 |
| Figura 62 Resumen de configuración..... | 110 |
| Figura 63 Vaciado de archivos | 110 |
| Figura 64 Fin de instalación | 111 |
| Figura 65 Ejecución del programa..... | 111 |
| Figura 66 Programa en el panel de aplicaciones | 112 |
| Figura 67 Ruta de archivos | 112 |
| Figura 68 Panel de categorías, sin JAVA..... | 113 |
| Figura 69 Programa dentro del sistema | 114 |
| Figura 70 Entorno de edición | 115 |
| Figura 71 Acción Nuevo | 116 |
| Figura 72 Acción Abrir | 116 |
| Figura 73 Navegación de los resultados de compilación | 117 |
| Figura 74 Panel de la Tabla de símbolos | 118 |
| Figura 75 Panel de la generación del código optimizado | 119 |
| Figura 76 Contenedor del Análisis léxico | 120 |
| Figura 77 Contenedor del Análisis sintáctico | 121 |
| Figura 78 Contenedor del Análisis Semantico..... | 122 |
| Figura 79 Contenedor del Código Intermedio..... | 123 |
| Figura 80 Ventana de Créditos..... | 124 |
| Figura 81 Ventana de la Documentación | 125 |
| Figura 82 Carpeta de códigos ejemplos..... | 125 |

Capítulo I. Antecedentes del lenguaje

Nombre del lenguaje

Coffe-Code



Objetivo del lenguaje

El lenguaje va dirigido a personas que comienzan a aprender programación, ya que permitirá el aprendizaje de la construcción de la lógica, la creación de aplicaciones sencillas para acercarse a la lógica de un lenguaje de bajo nivel, y el entendimiento del propio lenguaje a través de un compilador.

Símbolos del lenguaje

- Alfabeto

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J |
| K | L | M | N | Ñ | O | P | Q | R | S |
| T | U | V | W | X | Y | Z | a | b | c |
| d | e | f | g | h | i | j | k | l | m |
| n | ñ | o | p | q | r | s | t | u | v |
| w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | = | + | - | / | * | ^ |
| % | > | < | ! | & | | (|) | { | } |
| # | _ | ; | , | . | | | | | |

- Tipo de dato

| |
|---------|
| int |
| float |
| String |
| boolean |

- Palabras reservadas

| |
|--------|
| printf |
| scanf |
| if |
| else |
| while |
| class |
| void |
| return |

- Operadores relacionales

| | |
|----|-------------------|
| = | Igual |
| < | Menor que |
| > | Mayor que |
| <= | Menor o igual que |
| >= | Mayor o igual que |

- Operadores aritméticos

| | |
|---|----------------|
| + | Suma |
| - | Resta |
| / | División |
| * | Multiplicación |
| ^ | Potencia |
| % | Residuo |

- Operadores lógicos

| | |
|----|---|
| && | Y |
| | O |

- Otros símbolos

| | |
|---|-----------------------|
| { | Llave que abre |
| } | Llave que cierra |
| (| Paracentesis que abre |
|) | Paréntesis que cierra |
| , | Coma |
| ; | Punto y coma |
| “ | Comilla doble |
| ‘ | Comilla simple |
| . | Punto |

- Separadores

| | |
|----|-------------------|
| \b | Espacio en blanco |
| \t | Tabulador |
| \r | Enter |
| \n | Linea nueva |

Capítulo II. Análisis léxico

Expresiones regulares

Para definir las expresiones regulares de las reglas léxicas, se utilizarán los siguientes conjuntos:

$L = \{A-Z, a-z\}$

$D = \{0-9\}$

- **Variables o Identificadores**

Inicia con letra (A-Z, a-z) le puede seguir letra o número (0-9) cero o más veces.

Expresión regular:

$L(L|D)^*$

Ejemplos:

Correctos: a, var2, PI, nomUsu123

Incorrectos: 1var, %nom, POO\$, 00NUMBER\$

- **Entero (int)**

Sera un dígito de (0-9) n (uno o más) veces.

Expresión regular:

D^+

Ejemplos:

Correctos: 0, 1, 10, 200, 9000000

Incorrectos: -1, -0, 0.0, 9.&, 99n

- **Flotante (float)**

Sera n dígitos después un punto seguido de n dígitos.

Expresión regular:

$D^+.D^+$

Ejemplos:

Correctos: 0.0, 1.0, 32234.0123, 999.3, 0.983664, 99999.0

Incorrectos: 1, 982, 873, 94f, 8763, .009883

- **Cadena (String)**

Inicia con comilla simple le sigue letra o dígito, cero o más veces y termina con comilla simple.

Expresión regular:

`"(L|D)*"`

Ejemplos:

Correctos: "HOLA MUNDO", "Cadena 999", 'H', "09", '3', '123', "", ''

Incorrectos: Hola "mu" ndo, ' 000, " 98 &%3 34"

- **Booleano (boolean)**

Tiene un valor true o false.

Expresión regular:

`true | false`

Ejemplos:

Correctos: true, false

Incorrectos: TRUE, True, Verdadero, FALSO, FALSE, False

- **Mensaje**

Inicia con comilla doble, seguido de letra o dígito una o más veces y termina con comilla doble.

Expresión regular:

`"(L|D)+"`

Ejemplos:

Correctos: "3", "A", "HOLA MUNDO", "Hola mundo", "Mensaje 0012"

Incorrectos: Hola mundo', 'Mensaje 98 , ' , "

- **Clase**

Inicia con el símbolo de número, le sigue letra y le puede seguir letra o dígito cero o más veces.

Expresión regular:

`#L(L|D)*`

Ejemplos:

Correctos: `#Clase`, `#lexico`, `#m`, `#clase23`, `#Clase32nom`

Incorrectos: `#`, `#9883223`, `#C$$$`, `#Class"3"`, `#clase 21`

- **Método**

Inicia con un guion bajo, le sigue letra y le puede seguir letra o dígito cero o más veces.

Expresión regular:

`_L(L|D)*`

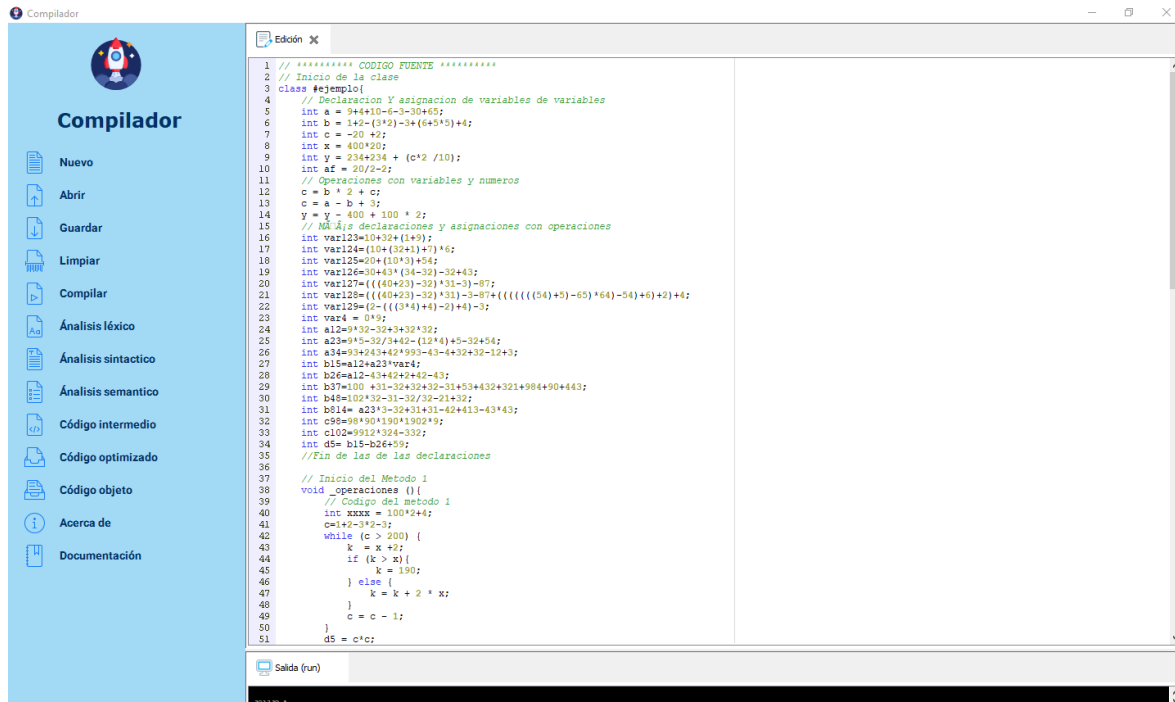
Ejemplos:

Correctos: `_metodo1`, `_m2`, `_limpiaCajas`, `_iteraVariable2`

Incorrectos: `método_metodo`, `_2`, `_`, `_32metodo`, `_$clase`

Resultados del análisis léxico

En la Figura 1, se muestran el código fuente a ejecutar.



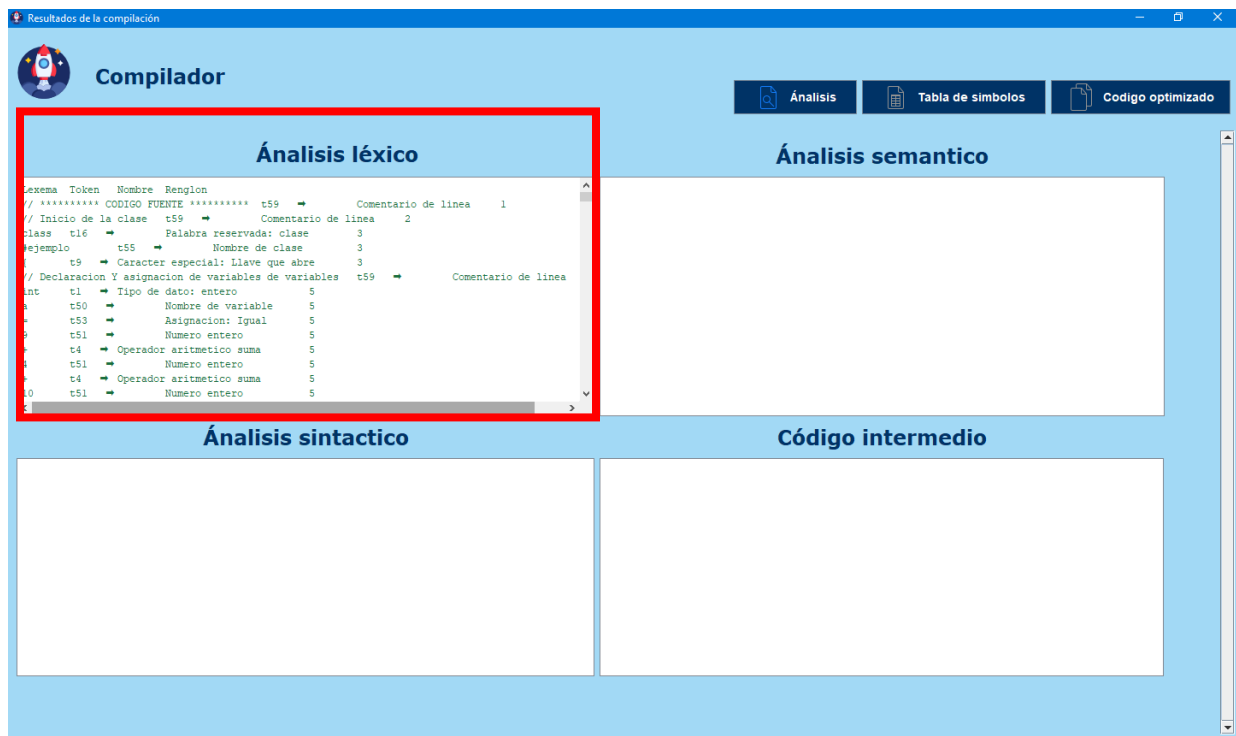
```

1 // ***** CODIGO FUENTE *****
2 // Inicio de la clase
3 class #ejemplo{
4 // Declaracion Y asignacion de variables de variables
5 int a = 944+10-6-3-30+65;
6 int b = 1+2-(3*2)-3+(6+5*5)+4;
7 int c = -20 +2;
8 int x = 400*20;
9 int y = 234+234 + (c*2 /10);
10 int af = 20/2-2;
11 // Operaciones con variables y numeros
12 c = b + 2 + c;
13 c = a - b + 3;
14 y = y - 400 + 100 * 2;
15 // Muestra declaraciones y asignaciones con operaciones
16 int var123=(10+32+(1+9));
17 int var124=(10+(32+1)+7)*6;
18 int var125=20+(10*3)+54;
19 int var126=30+43*(34-32)-32+43;
20 int var127=((40+23)-32)*31-3-87;
21 int var128=((40+23)-32)*31-3-87+((((54)+5)-65)*64)-54+6)+2)+4;
22 int var129=(2-((3*4)+4)-2)*4-3;
23 int var4 = 0*9;
24 int a12=9*32-32+3+32*32;
25 int a23=9*5-32/3+42-(12*4)+5-32+54;
26 int a34=9+2+3+42+993-43-4+32+32-12+3;
27 int b15=a12+a23*var4;
28 int b26=a12-43+42+2+42-43;
29 int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
30 int b48=100*50-31-32/32-21+32;
31 int b814= a23*3-32+31+31-42+413-43*43;
32 int c98=98*90+190*1902*9;
33 int c102=9912*324-332;
34 int d5= b15-b26+9;
35 //Fin de las de las declaraciones
36
37 // Inicio del Metodo 1
38 void _operaciones () {
39 //Codigo del metodo 1
40 int xxx = 100*2+4;
41 c=1+2-3*2-3;
42 while (c > 200) {
43 k = x +2;
44 if (k > x) {
45 k = 130;
46 } else {
47 k = k + 2 * x;
48 }
49 c = c - 1;
50 }
51 d5 = c*c;

```

Figura 1 Código fuente del léxico

En la Figura 2, se muestra el análisis léxico.



| Lexema | Token | Nombre | Renglon |
|--|-------|-----------------------------------|---------|
| // ***** CODIGO FUENTE ***** | t59 | Comentario de linea | 1 |
| // Inicio de la clase | t59 | Comentario de linea | 2 |
| class | t16 | Palabra reservada: class | 3 |
| #ejemplo | t55 | Nombre de clase | 3 |
| { | t9 | Caracter especial: Llave que abre | 3 |
| // Declaracion Y asignacion de variables de variables | t59 | Comentario de linea | |
| int | t1 | Tipo de dato: entero | 5 |
| a | t50 | Nombre de variable | 5 |
| = | t53 | Asignacion: Igual | 5 |
| 944+10-6-3-30+65; | t51 | Numero entero | 5 |
| b | t4 | Operador aritmetico suma | 5 |
| = | t51 | Numero entero | 5 |
| 1+2-(3*2)-3+(6+5*5)+4; | t4 | Operador aritmetico suma | 5 |
| c | t51 | Numero entero | 5 |
| = | t51 | Numero entero | 5 |
| -20 +2; | | | |
| x | | | |
| = | | | |
| 400*20; | | | |
| y | | | |
| = | | | |
| 234+234 + (c*2 /10); | | | |
| af | | | |
| = | | | |
| 20/2-2; | | | |
| // Operaciones con variables y numeros | | | |
| c | | | |
| = | | | |
| b + 2 + c; | | | |
| c | | | |
| = | | | |
| a - b + 3; | | | |
| y | | | |
| = | | | |
| y - 400 + 100 * 2; | | | |
| // Muestra declaraciones y asignaciones con operaciones | | | |
| int | | | |
| var123=(10+32+(1+9)); | | | |
| int | | | |
| var124=(10+(32+1)+7)*6; | | | |
| int | | | |
| var125=20+(10*3)+54; | | | |
| int | | | |
| var126=30+43*(34-32)-32+43; | | | |
| int | | | |
| var127=((40+23)-32)*31-3-87; | | | |
| int | | | |
| var128=((40+23)-32)*31-3-87+((((54)+5)-65)*64)-54+6)+2)+4; | | | |
| int | | | |
| var129=(2-((3*4)+4)-2)*4-3; | | | |
| int | | | |
| var4 = 0*9; | | | |
| int | | | |
| a12=9*32-32+3+32*32; | | | |
| int | | | |
| a23=9*5-32/3+42-(12*4)+5-32+54; | | | |
| int | | | |
| a34=9+2+3+42+993-43-4+32+32-12+3; | | | |
| int | | | |
| b15=a12+a23*var4; | | | |
| int | | | |
| b26=a12-43+42+2+42-43; | | | |
| int | | | |
| b37=100 +31-32+32+32-31+53+432+321+984+90+443; | | | |
| int | | | |
| b48=100*50-31-32/32-21+32; | | | |
| int | | | |
| b814= a23*3-32+31+31-42+413-43*43; | | | |
| int | | | |
| c98=98*90+190*1902*9; | | | |
| int | | | |
| c102=9912*324-332; | | | |
| int | | | |
| d5= b15-b26+9; | | | |
| //Fin de las de las declaraciones | | | |
| | | | |
| // Inicio del Metodo 1 | | | |
| void _operaciones () { | | | |
| //Codigo del metodo 1 | | | |
| int | | | |
| xxx = 100*2+4; | | | |
| c=1+2-3*2-3; | | | |
| while (c > 200) { | | | |
| k = x +2; | | | |
| if (k > x) { | | | |
| k = 130; | | | |
| } else { | | | |
| k = k + 2 * x; | | | |
| } | | | |
| c = c - 1; | | | |
| } | | | |
| d5 = c*c; | | | |

Figura 2 Resultado del análisis léxico

Capítulo III. Análisis sintáctico

Gramáticas descritas y ejemplos

- **Principal**

Comienza con la palabra class, le sigue nombre de la clase, le sigue llave que abre le pueden seguir declaraciones y métodos o solo métodos y termina con llave que cierra.

<INICIO> → <INICIO1> | <INICIO2>

<INICIO1> → Class Inicio Llave_a <METODOS> Llave_c

<INICIO2> → Class Inicio Llave_a <DECLARACIONES> <METODOS> Llave_c

Ejemplo:

```
class #nomClase {  
    int var1=32;  
    void _metodo(){  
        //Instrucciones  
    }  
}
```

- **Métodos**

Puede ser uno método o más de uno.

<METODOS> → <METODO1>

<METODO1> → <METODO> <METODO2>

<METODO2> → <METODO1> | λ

Ejemplo:

```
void _metodo1() {  
}  
Int _metodo2(){  
}
```

- **Método**

Puede iniciar con un tipo de dato seguido del nombre del método, un paréntesis que abre, un paréntesis que cierra, una llave que abre, una **sentencia**, un return, un **valor**, un punto y coma y termina con llave que cierra o puede iniciar con void seguido del nombre del método, un paréntesis que abre, un paréntesis que cierra, le sigue **sentencia** y termina con llave que cierra

<METODO> → <MET1> | <MET2>

<MET1> → tipo de dato Nombre del Método Parentesis_a Parentesis_c Llave_a
<SENTENCIA> Return <VALOR> P_coma Llave_c

<MET2> → Void Metodo Parentesis_a Parentesis_c Llave_a <SENTENCIA>
Llave_c

Ejemplos:

```
String _metodoReturn( ){  
    String s= "Hola mundo";  
    //Sentencias  
    return s;  
}  
  
Void _metodo() {  
    //Sentencias  
}
```

- **Sentencia**

Puede ser una **declaración** o una **asignación** o una **condición** o **sentencia de repetición** o **lectura** o **escritura**, se puede repetir una o más veces

<SENTENCIA> → <SENTENCIA1>

<SENTENCIA1> → <DECLARACION> |

<ASIGNACION> |

<IF> |

<WHILE> |

<LECTURA> |

<ESCRITURA> |

<SENTENCIA>

Ejemplos:

Sentencias:

```
if( <condición> ) {  
    //Sentencias  
}  
  
While( <condición> ) {  
    //Sentencias  
}
```

- **Declaración**

Inicia con el tipo de dato, le sigue el nombre de la variable, le puede seguir signo igual seguido de un **valor** o una operación matemática no y termina con punto y coma.

Ejemplos:

```
String var1; //Solo se declara sin inicializarla  
  
String var1 = "VALORES"; // se le asigna una cadena  
  
int var2 = (3 + 4); //Operación matemática  
  
int var2 = (3 + 4) * (32 / 43) + (4^2); //Operación matemática
```

- **Valor**

Puede ser un entero, flotante, variable o carácter

<VALOR> → Numero | Cadena | Flotante | variable

Ejemplos:

```
Valor= 0  
  
Valor = "Hola"  
  
Valor = 0.0  
  
Valor = var1
```

- **Números**

Es un entero o flotante

<NUMEROS> → Numero | Flotante

Ejemplo:

```
Numero = 0 , 1, 2000
```

Numero = 0.0 , 20.0, 32.1, 0.32

- **Booleanos**

Es un true o false

<BOOLEANOS> → True | False

Ejemplos:

Booleano = true o false

- **Comparación**

Inicia con una operación relacional, se le puede agregar un operador lógico seguido de agrega comparación cero o más veces.

<COMPARACION> → <OPERACIÓN_REL> <COMPARACION1>

<COMPARACION1> → <AGREGA_COMPARACION> | ^

<AGREGA_COMPARACION> → Op_logico <COMPARACION>

Ejemplo:

Comparación= (92 + 32) > (32)

Comparación = (43 > 9) && (2 > 43)

Comparación = ((43+32) > 9) && (2 > 43)

- **Operación relacional**

Puede iniciar con un valor o una operación matemática, le sigue un operador relacional y le puede seguir un valor o una operación matemática.

Inicia con una operación relacional, se le puede agregar un operador lógico seguido de otra comparación una o más veces.

<OPERACION_REL> → <VALOR> Op_relacional <VALOR> |

<OPERACION_MAT> Op_relacional VALOR |

<VALOR> Op_relacional <OPERACION_MAT> |

<OPERACION_MAT> Op_relacional <OPERACION_MAT>

Ejemplos:

Operación relacional = valor > valor

Operación relacional = valor < valor

Operación relacional = valor >= valor

Operación relacional = valor <= valor

Operación relacional = valor != valor

Operación relacional = valor == valor

Operación relacional = (valor + valor) > (valor * valor + valor)

- **Operación lógica**

Inicia con una operación lógica, seguido de **agrega comparación** una o más veces.

<OPERACION_LOG> → <OPERACION_LOG1>

<OPERACION_LOG1> → OPERACION_REL AGREGA_COMPARACION
<OPERACION_LOG>

Ejemplos:

Operación lógica = (Operación lógica) && (Operación relacional)

Operación lógica = (Operación lógica) && (Operación relacional) || (operación lógica)

Operación lógica = (Operación lógica) && (Operación relacional && operación relacional)

- **Condición**

Inicia con la palabra **if** le sigue (**comparación**) le sigue { **sentencias** } le puede seguir **else** { **sentencias** }

<IF> → If Parentesis_a <COMPARACION> Parentesis_c Llave_a <SENTENCIA>
Llave_c <IF_ELSE>

<IF_ELSE> → Else Llave_a <SENTENCIA> Llave_c | λ

Ejemplos:

```
if(true){  
    //Sentencias
```

```
}
```

```
if(var > var){  
    //Sentencia
```

```
}else {  
    //Sentencia
```

```
}  
If( Operación relacional ){  
    //Sentencias  
}  
If( operaciones logicas) {  
    //Sentencias  
}
```

- **Sentencias de repetición**

Inicia con la palabra while seguido de paréntesis que abre seguido de **comparación** seguido de paréntesis que cierra seguido de llave que abre seguido de **sentencia** termina con llave que cierra.

WHILE → While Parentesis_a COMPARACION Parentesis_c Llave_a
SENTENCIA Llave_c

Ejemplo:

```
While( comparación ){  
    //Sentencias  
}  
While( operación lógica ){  
    //Sentencias  
}  
While( operación relacional ){  
    //Sentencias  
}
```

- **Lectura**

Inicia con la palabra scanf, sigue paréntesis que abre, le sigue una variable, después el paréntesis que cierra y termina con punto y coma.

<LECTURA> → scanf Parentesis_a Variable Parentesis_c P_coma

Ejemplo:

scanf (variable) ;

- **Escritura**

Inicia con la palabra printf sigue un paréntesis que abre, el mensaje, puede seguir coma seguido de un **valor** en seguida el paréntesis que cierra y termina con punto y coma.

<ESCRITURA> → printf Parentesis_a VALOR Parentesis_c P_coma

Ejemplo:

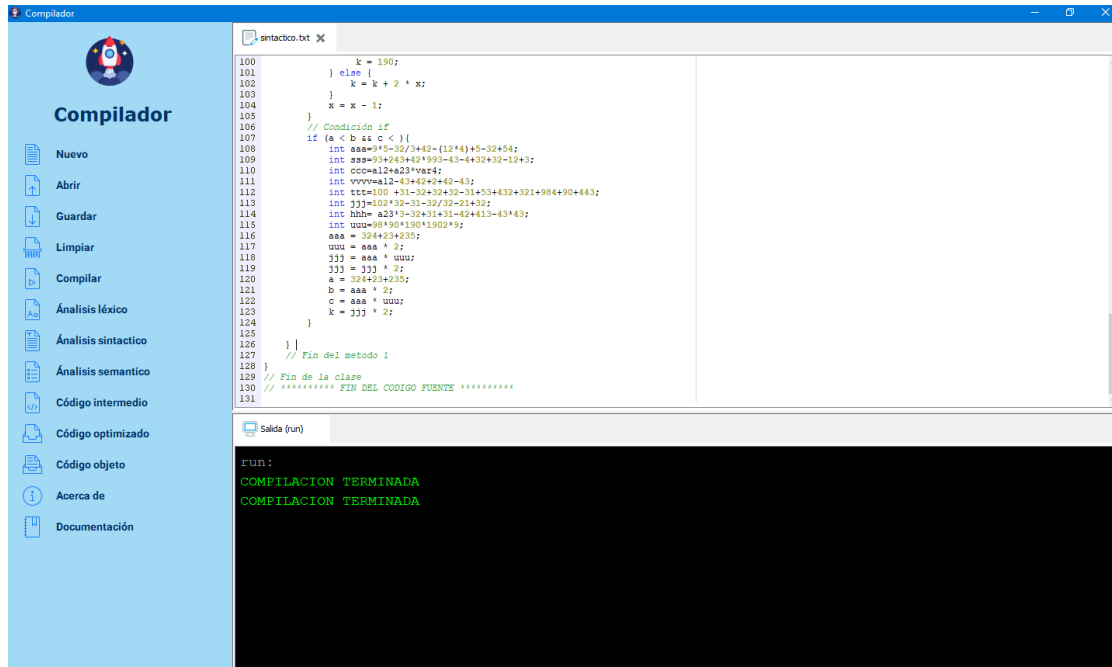
Printf(valor);

Printf("Hola mundo ");

Printf(43);

Resultados del análisis sintáctico

En la Figura 3, el código fuente a analizar sintácticamente.



```

100      k = 190;
101    } else {
102      k = k + 2 * x;
103    }
104    x = x - 1;
105  }
106  // Condición if
107  if (a < b && c < ) {
108    int aaa=95+32/34+2-((12*4)+5-32+54;
109    int sss=33+245+40*993-43-4+32+32-12+3;
110    int ccc=a12+23*var4;
111    int vvvv=a12-43+42+42-43;
112    int ttt=100 +31-32+32+32-31+59+432+321+984+90+443;
113    int jjj=102*32-31-32/32-21+32;
114    int hhh= a23*5-32+31+31-42+413-43*43;
115    int uuu=90*90*100*1002*9;
116    aaa = 324+23+235;
117    uuu = aaa * 2;
118    jjj = aaa * uuu;
119    jjj = jjj * 2;
120    a = 324+23+235;
121    b = aaa * 2;
122    c = aaa * uuu;
123    k = jjj * 2;
124  }
125  }
126  } // Fin del metodo 1
127  // Fin de la clase
128  // ***** FIN DEL CODIGO FUENTE *****
129
130
131

```

Salida (run)

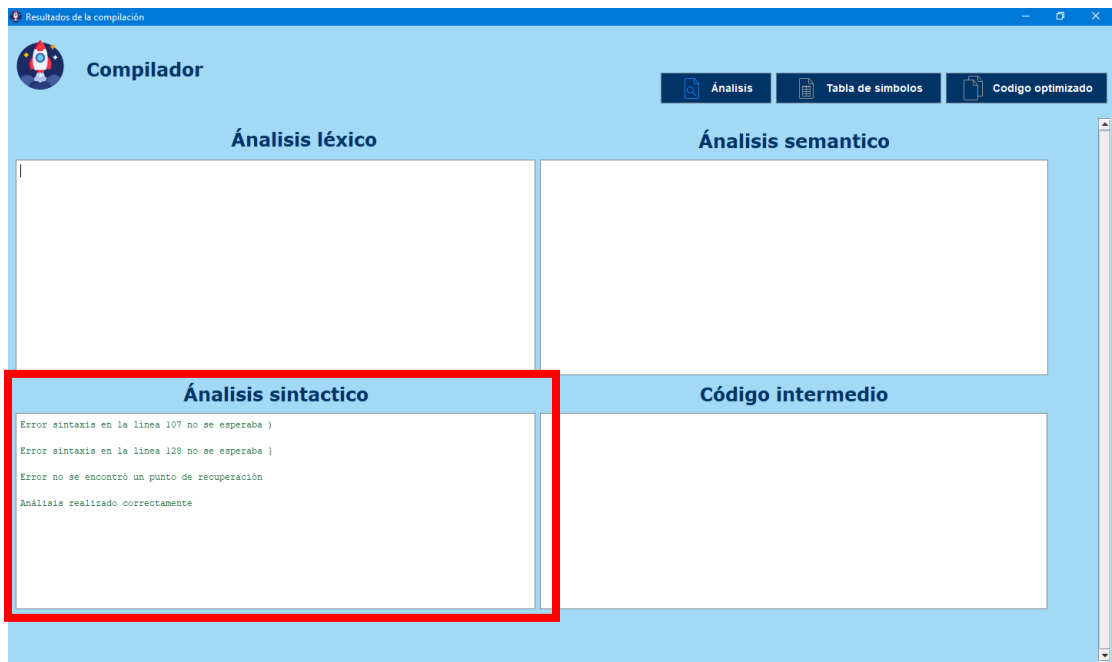
```

RUN:
COMPILACION TERMINADA
COMPILACION TERMINADA

```

Figura 3 Código fuente del análisis sintáctico

En la Figura 4, el resultado del análisis sintáctico.



Resultados de la compilación

Compilador

Álisis Tabla de símbolos Código optimizado

Álisis léxico Álisis semantico

Álisis sintactico Código intermedio

```

Error sintaxis en la línea 107 no se esperaba )
Error sintaxis en la línea 128 no se esperaba }
Error no se encontró un punto de recuperación
Análisis realizado correctamente

```

Figura 4 Resultado del análisis sintáctico

Capítulo IV. Análisis semántico

Validación semántica implementada

Diseño de tablas, para evaluar las variables

| Tabla de símbolos | | | | | |
|-------------------|-----------|-------|--------------|-------|-----------|
| Lexema | No. Token | Valor | Tipo de dato | Usada | Declarada |
| | | | | | |
| | | | | | |

| Tabla de prioridades | |
|----------------------|--|
| () | Paréntesis |
| ^ | Potencia |
| / * % | Multiplicación, división |
| + - | Suma, resta |
| >, <, <=, >=, ==, != | Operadores Relacionales y de comparación |
| &&, | Operadores Lógicos |

Tablas de resultados

| SUMA | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Entero |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| RESTA | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Entero |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| MULTIPLICACIÓN | | RESULTADO |
|----------------|----------|-----------|
| Entero | Entero | Entero |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| DIVISIÓN | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Flotante |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| MODULO | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Entero |
| Flotante | Entero | Entero |
| Entero | Flotante | Entero |
| Flotante | Flotante | Entero |

| POTENCIA | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Flotante |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| DIVISIÓN | | RESULTADO |
|----------|----------|-----------|
| Entero | Entero | Flotante |
| Flotante | Entero | Flotante |
| Entero | Flotante | Flotante |
| Flotante | Flotante | Flotante |

| OPERADOR LOGICA | | RESULTADO |
|-----------------|---------|-----------|
| Boolean | Boolean | Boolean |

| ASIGNACIÓN | | RESULTADO |
|------------|----------|-----------|
| Entero | Entero | true |
| Flotante | Entero | true |
| Entero | Flotante | false |
| Flotante | Flotante | true |
| Entero | String | false |
| Entero | Boolean | false |
| Flotante | String | false |
| Flotante | Boolean | false |
| Boolean | Entero | false |
| Boolean | Flotante | false |
| Boolean | String | false |
| Boolean | Boolean | true |
| String | Entero | false |
| String | Flotante | false |
| String | Boolean | false |
| String | String | true |

| OPERADOR RELACIONAL | | RESULTADO |
|---------------------|----------|-----------|
| Entero | Entero | True |
| Flotante | Entero | True |
| Entero | Flotante | True |
| Flotante | Flotante | True |
| String | Entero | False |
| String | Flotante | False |
| String | Boolean | False |
| String | String | True |
| Boolean | Entero | False |
| Boolean | Flotante | False |
| Boolean | Boolean | True |
| Boolean | String | False |

| Tabla de errores |
|--------------------------|
| Variable no declarada |
| Variable ya definida |
| Carácter no esperado |
| Variable no inicializada |
| Asignación no valida |
| Error de tipos |

Métodos utilizados

```
/**
 * Método que realiza la conversión de una notación infija, a una postfija.
 * Con el uso de pilas
 *
 * @param cola cadena resultado
 * @return una cola con la notación posfija
 */
public ColaD posfijo(ColaD cola) {

    ColaD pResultado = new ColaD();
    PilaD pOperadores = new PilaD();
    while (cola.getF() != null) {
        String s = cola.elimina(null).getS();
        if (isOperadorS(s)) {
            if (isOperador(s)) {
                if (pOperadores.getTope() != null) {
                    String s2 = pOperadores.elimina(null).getS();
                    if (isOperador(s2)) {
                        if (prioridad(s2) >= prioridad(s)) {
                            pOperadores.inserta(new Nodo(s, -1), null);
                            pResultado.inserta(new Nodo(s2, -1), null);
                        } else if (prioridad(s2) < prioridad(s)) {
                            pOperadores.inserta(new Nodo(s2, -1), null);
                            pOperadores.inserta(new Nodo(s, -1), null);
                        } else {
                            pOperadores.inserta(new Nodo(s2, -1), null);
                            pOperadores.inserta(new Nodo(s, -1), null);
                        }
                    }
                } else {
                    pOperadores.inserta(new Nodo(s2, -1), null);
                }
            }
        }
    }
}
```



```
        pOperadores.inserta(new Nodo(s, -1), null);
    }
    } else {
        pOperadores.inserta(new Nodo(s, -1), null);
    }
    } else if (s.equals("(")) {
        pOperadores.inserta(new Nodo(s, -1), null);
    } else if (s.equals(")")) {
        while (pOperadores.getTope() != null) {
            if (pOperadores.getTope().getS().equals("(")) {
                pOperadores.elimina(null);
                break;
            } else {
                String op = pOperadores.elimina(null).getS();
                pResultado.inserta(new Nodo(op, -1), null);
            }
        }
    }
    } else {
        pResultado.inserta(new Nodo(s, -1), null);
    }
}
while (pOperadores.getTope() != null) {
    String op = pOperadores.elimina(null).getS();
    pResultado.inserta(new Nodo(op, -1), null);
}

return pResultado;
}

/**
 * Método que verifica que el lexema sea un operador o no
 *
 * @param s
 * @return si es un operador, retorna true
 */
private boolean isOperador(String s) {
    return s.equals("%") || s.equals("**") || s.equals("^") || s.equals("/") ||
s.equals("+") || s.equals("-") || s.equals("!") || s.equals("&&") || s.equals("||") ||
s.equals("<") || s.equals(">") || s.equals(">=") || s.equals("<=") || s.equals("==") ||
s.equals("!=");
}
```

```
/**
 * Método que retorna el número de prioridad de las operaciones, segun su
 * prioridad
 *
 * @param operador
 * @return
 */
private int prioridad(String operador) {
    switch (operador) {
        case "^":
            return 6;
        case "/":
        case "*":
        case "%":
            return 5;
        case "+":
        case "-":
            return 4;
        case ">":
        case "<":
        case ">=":
        case "<=":
        case "==":
        case "!=":
            return 3;
        case "&&":
        case "||":
        case "!":
            return 2;
        default:
            break;
    }
    return 1;
}

/**
 * Metodo que realiza la operación, segun la prioridad de las operaciones
 *
 * @param pResultado
 * @return
 */
public Object operaciones(ColaD pResultado) {
    PilaD pOperacion = new PilaD();
```

```
Object resultado = null;
String res[];
while (pResultado.getF() != null) {
    String s = pResultado.elimina(null).getS();
    if (isOperador(s)) {
        String op2 = pOperacion.elimina(null).getS();

        String op1 = pOperacion.elimina(null).getS();

        String tipoOp1 = tipoDato(op1);
        String tipoOp2 = tipoDato(op2);

        if (!"NOT".equals(tablaResultados(tipoOp1, tipoOp2, s))) {

            res = expresionFinal(op1, op2, s, tipoOp1, tipoOp2);
            if ("TRUE".equals(res[0])) {
                pOperacion.inserta(new Nodo(String.valueOf(res[1]), -1), null);

                resultado = res[1];
            } else {
                //ERROR NO SE PUEDE REALIZAR LA OPERACION
                System.out.println(s);
                System.out.println("Error de operacion");
                break;
            }
        } else {
            //MARCAR UN ERROR DE INCOMPATIBILIDAD DE OPERACIONES
            System.out.println("Compatilibidad de operaciones");
            break;
        }
    } else {
        pOperacion.inserta(new Nodo(s, -1), null);
    }
}

return resultado;

}

/**
 * Método que regresa el valor resultante, comparando primero los tipos de
 * datos asociados, para realizar el parseo entre operandos La operación es
```

```
* binaria
*
* @param op1
* @param op2
* @param operador
* @param tipoDato1
* @param tipoDato2
* @return un arreglo de tipo String [0] - regresa si es valido hacer la
* operacion [1] - retorna el resultado final
*/
private String[] expresionFinal(String op1, String op2, String operador, String
tipoDato1, String tipoDato2) {
    String resultados[] = new String[3];

    if (tipoDato1.equals("INTEGER") && tipoDato2.equals("INTEGER")) {
        switch (operador) {
            case "+":
                try {
                    int var1 = Integer.parseInt(op1);
                    int var2 = Integer.parseInt(op2);
                    int res = var1 + var2;
                    resultados[1] = res + "";
                    resultados[0] = "TRUE";
                } catch (NumberFormatException e) {
                    resultados[0] = "FALSE";
                }
                break;
            case "-":
                try {
                    int var1 = Integer.parseInt(op1);
                    int var2 = Integer.parseInt(op2);
                    int var3 = var1 - var2;
                    resultados[1] = var3 + "";
                    resultados[0] = "TRUE";
                } catch (NumberFormatException e) {
                    resultados[0] = "FALSE";
                }
                break;
            case "*":
                try {
                    int var1 = Integer.parseInt(op1);
                    int var2 = Integer.parseInt(op2);
                    int var3 = var1 * var2;
```

```
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "/":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        int var3 = var2 / var1;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "^":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        int res = (int) Math.pow(var1, var2);

        resultados[1] = res + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "%":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        int var3 = var1 % var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case ">":
    try {
        int var1 = Integer.parseInt(op1);
```

```
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 > var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case ">=":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 >= var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "<":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 < var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "<=":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 <= var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "==":
    try {
```

```
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 == var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "!=":
    try {
        int var1 = Integer.parseInt(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 != var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "&&":
case "||":
    resultados[0] = "FALSE";
    break;
default:
    resultados[0] = "FALSE";
    break;
}
} else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
        case "+":
            try {
                int var1 = Integer.parseInt(op1);
                float var2 = Float.parseFloat(op2);
                float var3 = var1 + var2;
                resultados[1] = var3 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
            break;
        case "-":
            try {
```

```
int var1 = Integer.parseInt(op1);
float var2 = Float.parseFloat(op2);
float var3 = var1 - var2;
resultados[1] = var3 + "";
resultados[0] = "TRUE";
} catch (NumberFormatException e) {
    resultados[0] = "FALSE";
}
break;
case "*":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);
        float var3 = var1 * var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "/":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);
        float var3 = var1 / var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "^":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);
        String s = String.valueOf(Math.pow(var2, var1));
        float var3 = Float.parseFloat(s);
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
```



```
case "%":
  try {
    int var1 = Integer.parseInt(op1);
    float var2 = Float.parseFloat(op2);
    float var3 = var1 % var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case ">":
  try {
    int var1 = Integer.parseInt(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 > var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case ">=":
  try {
    int var1 = Integer.parseInt(op1);
    float var2 = Float.parseFloat(op2);

    boolean var3 = var1 >= var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case "<":
  try {
    int var1 = Integer.parseInt(op1);
    float var2 = Float.parseFloat(op2);

    boolean var3 = var1 < var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
```

```
        resultados[0] = "FALSE";
    }
    break;
case "<=":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);

        boolean var3 = var1 <= var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "==":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);

        boolean var3 = var1 == var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "!=":
    try {
        int var1 = Integer.parseInt(op1);
        float var2 = Float.parseFloat(op2);

        boolean var3 = var1 != var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "&&":
case "||":
    resultados[0] = "FALSE";
    break;
```

```
default:
    resultados[0] = "FALSE";
    break;
}

} else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("STRING")) {
    resultados[0] = "FALSE";
} else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("BOOLEAN")) {
    resultados[0] = "FALSE";
} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("INTEGER")) {
    switch (operador) {
        case "+":
            try {
                float var1 = Float.parseFloat(op1);
                int var2 = Integer.parseInt(op2);

                float var3 = var1 + var2;
                resultados[1] = var3 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
            break;
        case "-":
            try {
                float var1 = Float.parseFloat(op1);
                int var2 = Integer.parseInt(op2);
                float var3 = var1 - var2;
                resultados[1] = var3 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
            break;
        case "*":
            try {
                float var1 = Float.parseFloat(op1);
                int var2 = Integer.parseInt(op2);
                float var3 = var1 * var2;
                resultados[1] = var3 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
        }
    }
}
```

```
    }  
    break;  
case "/":  
    try {  
        float var1 = Float.parseFloat(op1);  
        int var2 = Integer.parseInt(op2);  
        float var3 = var1 / var2;  
        resultados[1] = var3 + "";  
        resultados[0] = "TRUE";  
    } catch (NumberFormatException e) {  
        resultados[0] = "FALSE";  
    }  
    break;  
case "^":  
    try {  
        float var1 = Float.parseFloat(op1);  
        int var2 = Integer.parseInt(op2);  
        String s = String.valueOf(Math.pow(var2, var1));  
        float var3 = Float.parseFloat(s);  
        resultados[1] = var3 + "";  
        resultados[0] = "TRUE";  
    } catch (NumberFormatException e) {  
        resultados[0] = "FALSE";  
    }  
    break;  
case "%":  
    try {  
        float var1 = Float.parseFloat(op1);  
        int var2 = Integer.parseInt(op2);  
        float var3 = var1 % var2;  
        resultados[1] = var3 + "";  
        resultados[0] = "TRUE";  
    } catch (NumberFormatException e) {  
        resultados[0] = "FALSE";  
    }  
    break;  
case ">":  
    try {  
        float var1 = Float.parseFloat(op1);  
        int var2 = Integer.parseInt(op2);  
        boolean var3 = var1 > var2;  
        resultados[1] = var3 + "";  
        resultados[0] = "TRUE";  
    }
```

```
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case ">=":
    try {
        float var1 = Float.parseFloat(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 >= var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "<":
    try {
        float var1 = Float.parseFloat(op1);
        int var2 = Integer.parseInt(op2);

        boolean var3 = var1 < var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "<=":
    try {
        float var1 = Float.parseFloat(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 <= var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "==":
    try {
        float var1 = Float.parseFloat(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 == var2;
```

```
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "!=":
    try {
        float var1 = Float.parseFloat(op1);
        int var2 = Integer.parseInt(op2);
        boolean var3 = var1 != var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "&&":
case "||":
    resultados[0] = "FALSE";
    break;
default:
    resultados[0] = "FALSE";
    break;
}

} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
        case "+":
            try {
                float var1 = Float.parseFloat(op1);
                float var2 = Float.parseFloat(op2);
                float var3 = var1 + var2;
                resultados[1] = var3 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
            break;
        case "-":
            try {
                float var1 = Float.parseFloat(op1);
                float var2 = Float.parseFloat(op2);
```

```
        float var3 = var1 - var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "*":
    try {
        float var1 = Float.parseFloat(op1);
        float var2 = Float.parseFloat(op2);
        float var3 = var1 * var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "/":
    try {
        float var1 = Float.parseFloat(op1);
        float var2 = Float.parseFloat(op2);
        float var3 = var1 / var2;
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "^":
    try {
        float var1 = Float.parseFloat(op1);
        float var2 = Float.parseFloat(op2);

        String s = String.valueOf(Math.pow(var1, var2));
        float var3 = Float.parseFloat(s);
        resultados[1] = var3 + "";
        resultados[0] = "TRUE";
    } catch (NumberFormatException e) {
        resultados[0] = "FALSE";
    }
    break;
case "%":
```

```
try {
  float var1 = Float.parseFloat(op1);
  float var2 = Float.parseFloat(op2);
  float var3 = var1 % var2;
  resultados[1] = var3 + "";
  resultados[0] = "TRUE";
} catch (NumberFormatException e) {
  resultados[0] = "FALSE";
}
break;
case ">":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 > var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case ">=":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 >= var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case "<":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 < var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
```



```
case "<=":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 <= var2;
    resultados[1] = var3 + "";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case "==":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 == var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case "!=":
  try {
    float var1 = Float.parseFloat(op1);
    float var2 = Float.parseFloat(op2);
    boolean var3 = var1 != var2;
    resultados[1] = var3 + "";
    resultados[0] = "TRUE";
  } catch (NumberFormatException e) {
    resultados[0] = "FALSE";
  }
  break;
case "&&":
case "||":
  resultados[0] = "FALSE";
  break;
default:
  resultados[0] = "FALSE";
  break;
}

} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("STRING")) {
  resultados[0] = "FALSE";
```

```
} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("BOOLEAN")) {
    resultados[0] = "FALSE";
} else if (tipoDato1.equals("STRING") && tipoDato2.equals("INTEGER")) {
    switch (operador) {
        case "+":
            try {
                String var1 = String.valueOf(op1);
                int var2 = Integer.parseInt(op2);
                resultados[1] = var1 + var2 + "";
                resultados[0] = "TRUE";
            } catch (NumberFormatException e) {
                resultados[0] = "FALSE";
            }
            break;
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
        case ">=":
        case "<":
        case "<=":
        case "==":
        case "!=":
        case "&&":
        case "||":
            resultados[0] = "FALSE";
            break;
        default:
            resultados[0] = "FALSE";
            break;
    }
}

} else if (tipoDato1.equals("STRING") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
        case "+":
            try {
                String var1 = String.valueOf(op1);
                float var2 = Float.parseFloat(op2);

                resultados[1] = var1 + var2 + "";
                resultados[0] = "TRUE";
            }
```

```
        } catch (NumberFormatException e) {
            resultados[0] = "FALSE";
        }
        break;
    case "-":
    case "*":
    case "/":
    case "^":
    case "%":
    case ">":
    case ">=":
    case "<":
    case "<=":
    case "==":
    case "!=":
    case "&&":
    case "||":
        resultados[0] = "FALSE";
        break;
    default:
        resultados[0] = "FALSE";
        break;
}

} else if (tipoDato1.equals("STRING") && tipoDato2.equals("STRING")) {
    switch (operador) {
        case "+":
            try {
                String var1 = String.valueOf(op1);
                String var2 = String.valueOf(op2);

                resultados[1] = var1 + var2 + "";
                resultados[0] = "TRUE";
            } catch (Exception e) {
                resultados[0] = "FALSE";
            }
            break;
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
```

```
case ">=":
case "<":
case "<=":
    resultados[0] = "FALSE";
    break;
case "==":
    try {
        String var1 = String.valueOf(op1);
        String var2 = String.valueOf(op2);

        resultados[1] = (var1.equals(var2)) + "";
        resultados[0] = "TRUE";
    } catch (Exception e) {
        resultados[0] = "FALSE";
    }
    break;
case "!=":
    try {
        String var1 = String.valueOf(op1);
        String var2 = String.valueOf(op2);

        resultados[1] = (!var1.equals(var2)) + "";
        resultados[0] = "TRUE";
    } catch (Exception e) {
        resultados[0] = "FALSE";
    }
    break;
case "&&":
case "||":
    resultados[0] = "FALSE";
    break;
default:
    resultados[0] = "FALSE";
    break;
}

} else if (tipoDato1.equals("STRING") && tipoDato2.equals("BOOLEAN")) {
    switch (operador) {
        case "+":
            try {
                String var1 = String.valueOf(op1);
                boolean var2 = Boolean.parseBoolean(op2);
```

```
        resultados[1] = (var1 + var2) + "";
        resultados[0] = "TRUE";
    } catch (Exception e) {
        resultados[0] = "FALSE";
    }
```

```
        break;
    case "-":
    case "*":
    case "/":
    case "^":
    case "%":
    case ">":
    case ">=":
    case "<":
    case "<=":
    case "==":
    case "!=":
    case "&&":
    case "||":
        resultados[0] = "FALSE";
        break;
    default:
        resultados[0] = "FALSE";
        break;
}
```

```
} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("INTEGER")) {
    switch (operador) {
        case "+":
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
        case ">=":
        case "<":
        case "<=":
        case "==":
        case "!=":
        case "&&":
        case "||":
```

```
        resultados[0] = "FALSE";
        break;
    default:
        resultados[0] = "FALSE";
        break;
    }

} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
        case "+":
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
        case ">=":
        case "<":
        case "<=":
        case "==":
        case "!=":
        case "&&":
        case "||":
            resultados[0] = "FALSE";
            break;
        default:
            resultados[0] = "FALSE";
            break;
    }
}

} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("STRING")) {
    switch (operador) {
        case "+":
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
        case ">=":
        case "<":
        case "<=":
        case "==":
```

```
case "!=":
case "&&":
case "||":
    resultados[0] = "FALSE";
    break;
default:
    resultados[0] = "FALSE";
    break;
}

} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("BOOLEAN")) {
    switch (operador) {
        case "+":
        case "-":
        case "*":
        case "/":
        case "^":
        case "%":
        case ">":
        case ">=":
        case "<":
        case "<=":
        case "==":
        case "!=":
            resultados[0] = "FALSE";
            break;
        case "&&":
            try {
                boolean var1 = Boolean.parseBoolean(op1);
                boolean var2 = Boolean.parseBoolean(op2);

                resultados[1] = (var1 && var2) + "";
                resultados[0] = "TRUE";
            } catch (Exception e) {
                resultados[0] = "FALSE";
            }
            break;
        case "||":
            try {
                boolean var1 = Boolean.parseBoolean(op1);
                boolean var2 = Boolean.parseBoolean(op2);

                resultados[1] = (var1 || var2) + "";
```

```
        resultados[0] = "TRUE";
    } catch (Exception e) {
        resultados[0] = "FALSE";
    }
    break;
default:
    resultados[0] = "FALSE";
    break;
}

} else {
    resultados[0] = "FALSE";
}

return resultados;
}

/**
 * Método que verifica que tipo de dato es el que tiene una expresión
 *
 * @param s
 * @return
 */
public static String tipoDato(String s) {

    try {
        int S1 = Integer.parseInt(s);
        return "INTEGER";
    } catch (NumberFormatException e2) {
        try {
            float S2 = Float.parseFloat(s);
            return "FLOAT";
        } catch (NumberFormatException e3) {
            if (s.equals("true") || s.equals("false")) {
                return "BOOLEAN";
            } else {
                return "STRING";
            }
        }
    }
}

/**
```



```
* Método que revisa la compatibilidad de los tipos de datos con el operando
* para retornar el tipo de dato que resultará
*
* @param tipoDato1
* @param tipoDato2
* @param operador
* @return
*/
private static String tablaResultados(String tipoDato1, String tipoDato2, String
operador) {
    String bandera = "BOOLEAN";
    if (tipoDato1.equals("INTEGER") && tipoDato2.equals("INTEGER")) {
        switch (operador) {
            case "+":
                return "INTEGER";
            case "-":
                return "INTEGER";
            case "*":
                return "INTEGER";
            case "/":
                return "FLOAT";
            case "%":
                return "INTEGER";
            case "^":
                return "INTEGER";
            case ">":
            case "<":
            case ">=":
            case "<=":
            case "!=":
            case "==":
                return "BOOLEAN";
            case "&&":
            case "||":
                return "NOT";
            default:
                return "NOT";
        }
    }

    } else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("FLOAT")) {
        switch (operador) {
            case "+":
                return "FLOAT";
```

```
    case "-":
      return "FLOAT";
    case "*":
      return "FLOAT";
    case "/":
      return "FLOAT";
    case "%":
      return "FLOAT";
    case "^":
      return "FLOAT";
    case ">":
    case "<":
    case ">=":
    case "<=":
    case "!=":
    case "==":
      return "BOOLEAN";
    case "&&":
    case "||":
      return "NOT";
    default:
      return "NOT";
  }

} else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("STRING")) {
  return "NOT";
} else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("BOOLEAN")) {
  return "NOT";
} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("INTEGER")) {
  switch (operador) {
    case "+":
      return "FLOAT";
    case "-":
      return "FLOAT";
    case "*":
      return "FLOAT";
    case "/":
      return "FLOAT";
    case "%":
      return "FLOAT";
    case "^":
      return "FLOAT";
    case ">":
```

```
        case "<":
        case ">=":
        case "<=":
        case "!=":
        case "==":
            return "BOOLEAN";
        case "&&":
        case "||":
            return "NOT";
        default:
            return "NOT";
    }
} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
        case "+":
            return "FLOAT";
        case "-":
            return "FLOAT";
        case "*":
            return "FLOAT";
        case "/":
            return "FLOAT";
        case "%":
            return "FLOAT";
        case "^":
            return "FLOAT";
        case ">":
        case "<":
        case ">=":
        case "<=":
        case "!=":
        case "==":
            return "BOOLEAN";
        case "&&":
        case "||":
            return "NOT";
        default:
            return "NOT";
    }
}

} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("STRING")) {
    return "NOT";
} else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("BOOLEAN")) {
```

```
    return "NOT";
  } else if (tipoDato1.equals("STRING") && tipoDato2.equals("INTEGER")) {
    switch (operador) {
      case "+":
        return "STRING";
      default:
        return "NOT";
    }
  } else if (tipoDato1.equals("STRING") && tipoDato2.equals("FLOAT")) {
    switch (operador) {
      case "+":
        return "STRING";
      default:
        return "NOT";
    }
  } else if (tipoDato1.equals("STRING") && tipoDato2.equals("STRING")) {
    switch (operador) {
      case "+":
        return "STRING";
      case "-":
      case "*":
      case "/":
      case "%":
      case "^":
      case ">":
      case "<":
      case ">=":
      case "<=":
        return "NOT";
      case "!=":
      case "==":
        return "BOOLEAN";
      case "&&":
      case "||":
        return "NOT";
      default:
        return "NOT";
    }
  } else if (tipoDato1.equals("STRING") && tipoDato2.equals("BOOLEAN")) {
    switch (operador) {
      case "+":
        return "STRING";
      default:
```

```
        return "NOT";
    }
} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("INTEGER")) {
    return "NOT";
} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("FLOAT")) {
    return "NOT";
} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("STRING")) {
    return "NOT";
} else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("BOOLEAN")) {
    switch (operador) {
        case "&&":
        case "||":
            return "BOOLEAN";
        default:
            return "NOT";
    }
}

return bandera;
}

/**
 * Método que verifica que a una variable se le pueda asignar el tipo de
 * dato, dependiendo si es compatible con su mismo tipo de dato de la
 * variable
 *
 * @param tipoDato1
 * @param tipoDato2
 * @return
 */
public static boolean asignacion(String tipoDato1, String tipoDato2) {
    if (tipoDato1.equals("INTEGER") && tipoDato2.equals("INTEGER")) {
        return true;
    } else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("FLOAT")) {
        return false;
    } else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("STRING")) {
        return false;
    } else if (tipoDato1.equals("INTEGER") && tipoDato2.equals("BOOLEAN")) {
        return false;
    } else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("INTEGER")) {
        return true;
    } else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("FLOAT")) {
        return true;
    }
}
```

```
    } else if (tipoDato1.equals("FLOAT") && tipoDato2.equals("STRING")) {
        return false;
    } else if (tipoDato1.equals("FLOTANTE") && tipoDato2.equals("BOOLEAN")) {
        return false;
    } else if (tipoDato1.equals("STRING") && tipoDato2.equals("INTEGER")) {
        return false;
    } else if (tipoDato1.equals("STRING") && tipoDato2.equals("FLOAT")) {
        return false;
    } else if (tipoDato1.equals("STRING") && tipoDato2.equals("STRING")) {
        return true;
    } else if (tipoDato1.equals("STRING") && tipoDato2.equals("BOOLEAN")) {
        return false;
    } else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("INTEGER")) {
        return false;
    } else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("FLOAT")) {
        return false;
    } else if (tipoDato1.equals("BOOLEAN") && tipoDato2.equals("STRING")) {
        return false;
    } else {
        return tipoDato1.equals("BOOLEAN") && tipoDato2.equals("BOOLEAN");
    }
}

/**
 * Método que verifica que el lexema sea un operador o no
 *
 * @param s
 * @return si es un operador, retorna true
 */
public boolean isOperadorS(String s) {
    return s.equals("%") || s.equals("**") || s.equals("^") || s.equals("/") ||
s.equals("+") || s.equals("-") || s.equals("!") || s.equals("&&") || s.equals("||") ||
s.equals("<") || s.equals(">") || s.equals(">=") || s.equals("<=") || s.equals("==") ||
s.equals("!=") || s.equals("(") || s.equals(")");
}

public static String tipoDatoID(String tipoD) {
    switch (tipoD) {
        case "int":
            return "INTEGER";
        case "float":
            return "FLOAT";
    }
}
```

```
        case "String":
            return "STRING";
        case "boolean":
            return "BOOLEAN";
        default:
            return "?";
    }
}

/**
 * Método para las validaciones semánticas
 */
public static String analizar(ArrayList<TablaSimbolos> lexema) {
    tablaSimbolos = new ArrayList<>();
    Boolean repite;
    String tipo;
    for (TablaSimbolos tokenActual : lexema) {
        switch (tokenActual.getNumToken()) {
            case VARIABLE:
                TablaSimbolos tokenAnterior = lastToken(lexema, tokenActual);
                TablaSimbolos tokenSiguiente = nextToken(lexema, tokenActual);
                switch (tokenAnterior.getNumToken()) {
                    case TIPO_DATO:
                        if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
                            tokenSiguiente = nextToken(lexema, tokenActual);
                            tipo = tokenAnterior.getLexema();
                            switch (tokenSiguiente.getNumToken()) {
                                case PUNTO_Y_COMA:
                                    if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
                                        agregarVarTabSimbolos(tablaSimbolos, tokenActual.getLexema(),
tokenAnterior.getLexema(), true, false, 1,null);
                                    } else {
                                        errorVariableDefinida(tokenActual);
                                        cambiarEstadoUnicaVar(tablaSimbolos, tokenActual.getLexema(),
false);
                                    }
                                }
                            break;
                        case IGUAL:
                            tokenSiguiente = nextToken(lexema, tokenSiguiente);
                            ArrayList<String> arr = new ArrayList<String>();
                            boolean agrega = true, hacer=true;
                            if (tokenSiguiente.getNumToken() == PUNTO_Y_COMA) {
                                agrega = false;
                                hacer = false;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
        errorCaracterInserperado(tokenSiguiente);
        break;
    } else if (tokenSiguiente.getNumToken() == COMA ){
        agrega = false;
        hacer = false;
        errorCaracterInserperado(tokenSiguiente);
        break;
    }
    TablaSimbolos t = null;
    while(agrega){
        if (tokenSiguiente.getNumToken() == VARIABLE) {
            // Validar si esta declarada
            if (!isDuplicated(tablaSimbolos, tokenSiguiente.getLexema())) {
                errorNoDeclarada(tokenSiguiente);
                hacer = false;
            } else {
                if (valor(tablaSimbolos, tokenSiguiente.lexema) == null) {
                    errorNoInicializada(tokenSiguiente);
                    hacer = false;
                } else {
                    arr.add(valor(tablaSimbolos, tokenSiguiente.lexema));
                }
            }
        } else {
            arr.add(tokenSiguiente.lexema);
        }
        t = tokenSiguiente;
        tokenSiguiente = nextToken(lexema, tokenSiguiente);
        if (tokenSiguiente.numToken == COMA ||
tokenSiguiente.numToken == PUNTO_Y_COMA) {
            agrega = false;
            if (t.getNumToken() == OPERADOR_ARITMETICO ||
t.getNumToken() == OPERADOR_LOGICO || t.getNumToken() ==
OPERADOR_RELACIONAL) {
                errorCaracterInserperado(t);
                hacer = false;
            }
        }
    }
    if (hacer) {
        String tipoDatoid = Semantico.tipoDatoid(tipo);
        if (tipoDatoid.equals("?")) {
        } else {

```



```

        String valor = Semantico.conversionArrayCola(arr); //RETORNA EL
VALOR DE LA OPERACION
        String tipoDatoValor = Semantico.tipoDato(valor); //RETORNA EL
TIPO DE DATO DEL VALOR
        if (Semantico.asignacion(tipoDatoID, tipoDatoValor)) { //VALIDA
EL TIPO DE DATO DEL IDENTIFICADOR CON EL DEL VALOR
            agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tokenAnterior.getLexema(), true, true, 1,valor);
        } else {
            if (tipoDatoValor.equals("STRING")) {
                tipoDatoValor = "String";
            } else if (tipoDatoValor.equals("INTEGER")) {
                tipoDatoValor = "int";
            } else if (tipoDatoValor.equals("FLOAT")) {
                tipoDatoValor = "float";
            } else if (tipoDatoValor.equals("BOOLEAN")) {
                tipoDatoValor = "boolean";
            }
            errorTipos(t, tipoDatoValor, tipo);
        }
    }
}
tokenActual = nextToken(lexema, t);
if (tokenActual.getNumToken() == COMA) {
    repite = true;
    while (repite){
        tokenActual = nextToken(lexema, tokenActual);
        switch (tokenActual.getNumToken()) {
            case VARIABLE:
                if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
                    tokenSiguiente = nextToken(lexema, tokenActual);
                    switch (tokenSiguiente.getNumToken()) {
                        case IGUAL:
                            tokenSiguiente = nextToken(lexema, tokenSiguiente);
                            arr = new ArrayList<String>();
                            agrega = true;
                            hacer=true;
                            if (tokenSiguiente.getNumToken() ==
PUNTO_Y_COMA) {
                                agrega = false;
                                hacer = false;
                                errorCaracterInserperado(tokenSiguiente);
                                break;
                            } else if (tokenSiguiente.getNumToken() == COMA ){

```

```

        agrega = false;
        hacer = false;
        errorCaracterInserperado(tokenSiguiente);
        break;
    }
    t = null;
    while(agrega){
        if (tokenSiguiente.getNumToken() == VARIABLE) {
            // Validar si esta declarada
            if (!isDuplicated(tablaSimbolos,
tokenSiguiente.getLexema())) {

                errorNoDeclarada(tokenSiguiente);
                hacer = false;
            } else {
                if (valor(tablaSimbolos, tokenSiguiente.lexema)
== null) {

                    errorNoInicializada(tokenSiguiente);
                    hacer = false;
                } else {
                    arr.add(valor(tablaSimbolos,
tokenSiguiente.lexema));

                }
            }
        } else {
            arr.add(tokenSiguiente.lexema);
        }
        t = tokenSiguiente;
        tokenSiguiente = nextToken(lexema,
tokenSiguiente);

        if (tokenSiguiente.numToken == COMA ||
tokenSiguiente.numToken == PUNTO_Y_COMA) {
            agrega = false;
            if (t.getNumToken() == OPERADOR_ARITMETICO
|| t.getNumToken() == OPERADOR_LOGICO || t.getNumToken() ==
OPERADOR_RELACIONAL) {

                errorCaracterInserperado(t);
                hacer = false;
            }
        }
    }
    if (hacer) {
        String tipoDatoID = Semantico.tipoDatoID(tipo);
        if (tipoDatoID.equals("?")) {

```

```
        } else {
            String valor =
Semantico.conversionArrayCola(arr); //RETORNA EL VALOR DE LA OPERACION
            String tipoDatoValor = Semantico.tipoDato(valor);
//RETORNA EL TIPO DE DATO DEL VALOR
            if (Semantico.asignacion(tipoDatoID,
tipoDatoValor)) { //VALIDA EL TIPO DE DATO DEL IDENTIFICADOR CON EL DEL VALOR
                agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tokenAnterior.getLexema(), true, true, 1,valor);
            } else {
                if (tipoDatoValor.equals("STRING")) {
                    tipoDatoValor = "String";
                } else if (tipoDatoValor.equals("INTEGER")) {
                    tipoDatoValor = "int";
                } else if (tipoDatoValor.equals("FLOAT")) {
                    tipoDatoValor = "float";
                } else if (tipoDatoValor.equals("BOOLEAN")) {
                    tipoDatoValor = "boolean";
                }
                errorTipos(t, tipoDatoValor, tipo);
            }
        }
    }
}
break;
default:
    agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tipo, true, false, 1,null);
    break;
}
}else{
    errorVariableDefinida(tokenActual);
    cambiarEstadoUnicaVar(tablaSimbolos,
tokenActual.getLexema(), false);
}
break;
case PUNTO_Y_COMA:
    repite = false;
    break;
}
}
} else if (tokenActual.getNumToken() == PUNTO_Y_COMA) {
    if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
//        agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tokenAnterior.getLexema(), true, false, 1,null);
```

```

    } else {
        errorVariableDefinida(tokenActual);
        cambiarEstadoUnicaVar(tablaSimbolos, tokenActual.getLexema(),
false);
    }
    } else {
        errorCaracterInserperado(tokenActual);
    }
    break;
    case COMA:
        if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
            agregarVarTabSimbolos(tablaSimbolos, tokenActual.getLexema(),
tokenAnterior.getLexema(), true, false, 1,null);
        } else {
            errorVariableDefinida(tokenActual);
            cambiarEstadoUnicaVar(tablaSimbolos, tokenActual.getLexema(),
false);
        }
        repite = true;
        while (repite){
            tokenActual = nextToken(lexema, tokenActual);
            switch (tokenActual.getNumToken()) {
                case VARIABLE:
                    if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
                        tokenSiguiente = nextToken(lexema, tokenActual);
                        switch (tokenSiguiente.getNumToken()) {
                            case IGUAL:
                                tokenSiguiente = nextToken(lexema, tokenSiguiente);
                                arr = new ArrayList<String>();
                                agrega = true;
                                hacer=true;
                                if (tokenSiguiente.getNumToken() == PUNTO_Y_COMA) {
                                    agrega = false;
                                    hacer = false;
                                    errorCaracterInserperado(tokenSiguiente);
                                    break;
                                } else if (tokenSiguiente.getNumToken() == COMA ){
                                    agrega = false;
                                    hacer = false;
                                    errorCaracterInserperado(tokenSiguiente);
                                    break;
                                }
                                t = null;
                                while(agrega){

```

```
        if (tokenSiguiente.getNumToken() == VARIABLE) {
            // Validar si esta declarada
            if (!isDuplicated(tablaSimbolos,
tokenSiguiente.getLexema())) {
                errorNoDeclarada(tokenSiguiente);
                hacer = false;
            } else {
                if (valor(tablaSimbolos, tokenSiguiente.lexema) ==
null) {
                    errorNoInicializada(tokenSiguiente);
                    hacer = false;
                } else {
                    arr.add(valor(tablaSimbolos,
tokenSiguiente.lexema));
                }
            }
        } else {
            arr.add(tokenSiguiente.lexema);
        }
        t = tokenSiguiente;
        tokenSiguiente = nextToken(lexema, tokenSiguiente);
        if (tokenSiguiente.numToken == COMA ||
tokenSiguiente.numToken == PUNTO_Y_COMA) {
            agrega = false;
            if (t.getNumToken() == OPERADOR_ARITMETICO ||
t.getNumToken() == OPERADOR_LOGICO || t.getNumToken() ==
OPERADOR_RELACIONAL) {
                errorCaracterInserperado(t);
                hacer = false;
            }
        }
    }
    if (hacer) {
        String tipoDatoID = Semantico.tipoDatoID(tipo);
        if (tipoDatoID.equals("?")) {
        } else {
            String valor = Semantico.conversionArrayCola(arr);
//RETORNA EL VALOR DE LA OPERACION
            String tipoDatoValor = Semantico.tipoDato(valor);
//RETORNA EL TIPO DE DATO DEL VALOR
            if (Semantico.asignacion(tipoDatoID, tipoDatoValor))
{ //VALIDA EL TIPO DE DATO DEL IDENTIFICADOR CON EL DEL VALOR
```

```

                                agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tokenAnterior.getLexema(), true, true, 1,valor);
                                } else {
                                    if (tipoDatoValor.equals("STRING")) {
                                        tipoDatoValor = "String";
                                    } else if (tipoDatoValor.equals("INTEGER")) {
                                        tipoDatoValor = "int";
                                    } else if (tipoDatoValor.equals("FLOAT")) {
                                        tipoDatoValor = "float";
                                    } else if (tipoDatoValor.equals("BOOLEAN")) {
                                        tipoDatoValor = "boolean";
                                    }
                                    errorTipos(t, tipoDatoValor, tipo);
                                }
                            }
                        }
                    break;
                    default:
                        agregarVarTabSimbolos(tablaSimbolos,
tokenActual.getLexema(), tipo, true, false, 1,null);
                        break;
                    }
                } else {
                    errorVariableDefinida(tokenActual);
                    cambiarEstadoUnicaVar(tablaSimbolos,
tokenActual.getLexema(), false);
                }
                break;
                case PUNTO_Y_COMA:
                    repite = false;
                    break;
            }
        }
        break;
    }
} else {
    errorVariableDefinida(tokenActual);
}
break;
case PUNTO_Y_COMA: // Asignaciones
    tipo = tipo(tablaSimbolos, tokenActual.lexema);
    String var = tokenActual.lexema;
    boolean revisa = true, hacer = true;
    if (!isDuplicated(tablaSimbolos, var)) {

```

```
errorNoDeclarada(tokenActual);
break;
} else { // Actualizar valor
    tokenActual = nextToken(lexema, tokenActual);
    switch(tokenActual.getNumToken()){
        case IGUAL:
            tokenActual = nextToken(lexema, tokenActual);
            ArrayList<String> arr = new ArrayList<String>();
            TablaSimbolos t = null ;
            boolean agrega = true;
            switch(tokenActual.getNumToken()){
                case VARIABLE:
                    if (!isDuplicated(tablaSimbolos, tokenActual.getLexema())) {
                        errorNoDeclarada(tokenActual);
                        agrega = false;
                        hacer = false;
                    } else { // Actualizar valor
                        if (valor(tablaSimbolos, tokenActual.lexema) == null) {
                            errorNoInicializada(tokenActual);
                            agrega = false;
                            hacer = false;
                        } else {
                            arr.add(valor(tablaSimbolos, tokenActual.lexema));
                            tokenSiguiente = nextToken(lexema, tokenActual);
                            if (tokenSiguiente.numToken == COMA ||
tokenSiguiente.numToken == PUNTO_Y_COMA) {
                                agrega = false;
                                t = tokenActual;
                            }
                            while(agrega){
                                if (tokenSiguiente.getNumToken() == VARIABLE) {
                                    // Validar si esta declarada
                                    if (!isDuplicated(tablaSimbolos,
tokenSiguiente.getLexema())) {
                                        errorNoDeclarada(tokenSiguiente);
                                        hacer = false;
                                        hacer = false;
                                    } else {
                                        if (valor(tablaSimbolos, tokenActual.lexema) == null) {
                                            errorNoInicializada(tokenSiguiente);
                                            hacer = false;
                                            hacer = false;
                                        } else {
                                            arr.add(valor(tablaSimbolos, tokenActual.lexema));
```

```

        }
    }
    } else {
        arr.add(tokenSiguiente.lexema);
    }
    t = tokenSiguiente;
    tokenSiguiente = nextToken(lexema, tokenSiguiente);
    if (tokenSiguiente.numToken == COMA ||
tokenSiguiente.numToken == PUNTO_Y_COMA) {
        agrega = false;
        if (t.getNumToken() == OPERADOR_ARITMETICO ||
t.getNumToken() == OPERADOR_LOGICO || t.getNumToken() ==
OPERADOR_RELACIONAL) {
            errorCaracterInserperado(t);
            hacer = false;
        }
    }
    }
    if (hacer) {
        String tipoDatoID = Semantico.tipoDatoID(tipo);
        if (tipoDatoID.equals("?")) {
        } else {
            String valor = Semantico.conversionArrayCola(arr);
//RETORNA EL VALOR DE LA OPERACION
            String tipoDatoValor = Semantico.tipoDato(valor);
//RETORNA EL TIPO DE DATO DEL VALOR
            if (Semantico.asignacion(tipoDatoID, tipoDatoValor)) {
//VALIDA EL TIPO DE DATO DEL IDENTIFICADOR CON EL DEL VALOR
                cambiarValor(tablaSimbolos, var, valor);
                cambiarInicializada(tablaSimbolos, var, true);
            } else {
                if (tipoDatoValor.equals("STRING")) {
                    tipoDatoValor = "String";
                } else if (tipoDatoValor.equals("INTEGER")) {
                    tipoDatoValor = "int";
                } else if (tipoDatoValor.equals("FLOAT")) {
                    tipoDatoValor = "float";
                } else if (tipoDatoValor.equals("BOOLEAN")) {
                    tipoDatoValor = "boolean";
                }
                errorTipos(t, tipoDatoValor, tipo);
            }
        }
    }
}

```



```

        }
    }
    break;
}
break;
}
break;
}
break;
}
}
return mensajesError;
}

/**
 * Métodos de los errores semánticos
 */
private static void errorVariableDefinida(TablaSimbolos tokenActual){
    mensajesError += "Error en la línea " + tokenActual.getNumLinea() + ": La variable <" +
tokenActual.getLexema() + "> ya se encuentra definida\n";
}

private static void errorCaracterInserperado(TablaSimbolos tokenSiguiente){
    mensajesError += "Error en la línea " + tokenSiguiente.getNumLinea() + ": No se
esperaba <" + tokenSiguiente.getLexema() + ">\n";
}

private static void errorNoDeclarada(TablaSimbolos tokenSiguiente){
    mensajesError += "Error en la línea " + tokenSiguiente.getNumLinea() + ": La variable
<" + tokenSiguiente.getLexema() + "> no se encuentra declarada\n";
}

private static void errorNoInicializada(TablaSimbolos tokenSiguiente){
    mensajesError += "Error en la línea " + tokenSiguiente.getNumLinea() + ": La variable
<" + tokenSiguiente.getLexema() + "> no se encuentra inicializada\n";
}

private static void errorTipos(TablaSimbolos t , String tipoDatoValor, String tipo1){
    mensajesError += "Error en la línea " + t.getNumLinea() + ": <" + tipoDatoValor + "> no
compatible con <" + tipo1 + ">\n";
}

```

Resultado de la validación semántica

En la Figura 5, el código fuente a analizar semánticamente

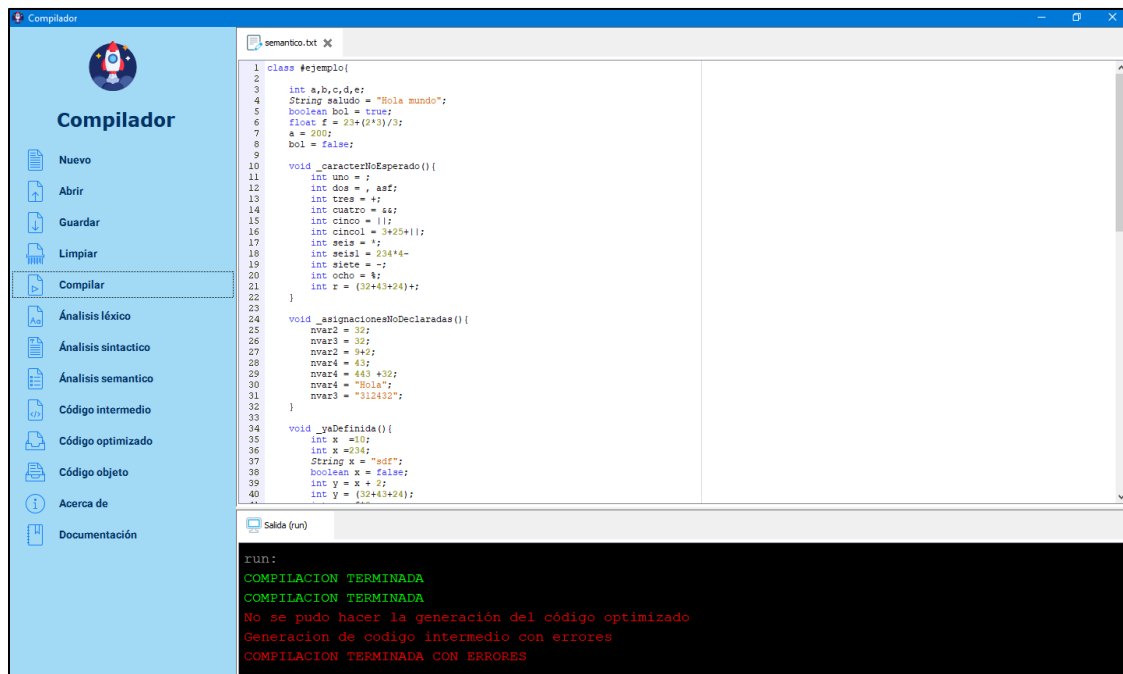


Figura 5 Ejecución del código fuente para el análisis semántico con errores

En la Figura 6, la representación de la tabla de variables.



| Tipo | Variable | Valor | Unica | Inicializada |
|---------|----------|--------------|-------|--------------|
| int | a | 200 | Si | Si |
| int | b | | Si | No |
| int | c | | Si | No |
| int | d | | Si | No |
| int | e | | Si | No |
| String | saludo | "Hola mundo" | Si | Si |
| boolean | bol | false | Si | Si |
| float | f | 25 | Si | Si |
| int | x | 10 | Si | Si |
| int | y | 12 | Si | Si |
| float | var | 21 | Si | Si |
| float | flotante | 41.4 | Si | Si |
| String | var3 | "Hola mundo" | Si | Si |
| int | af | 8 | Si | Si |

Figura 6 Tabla de símbolos

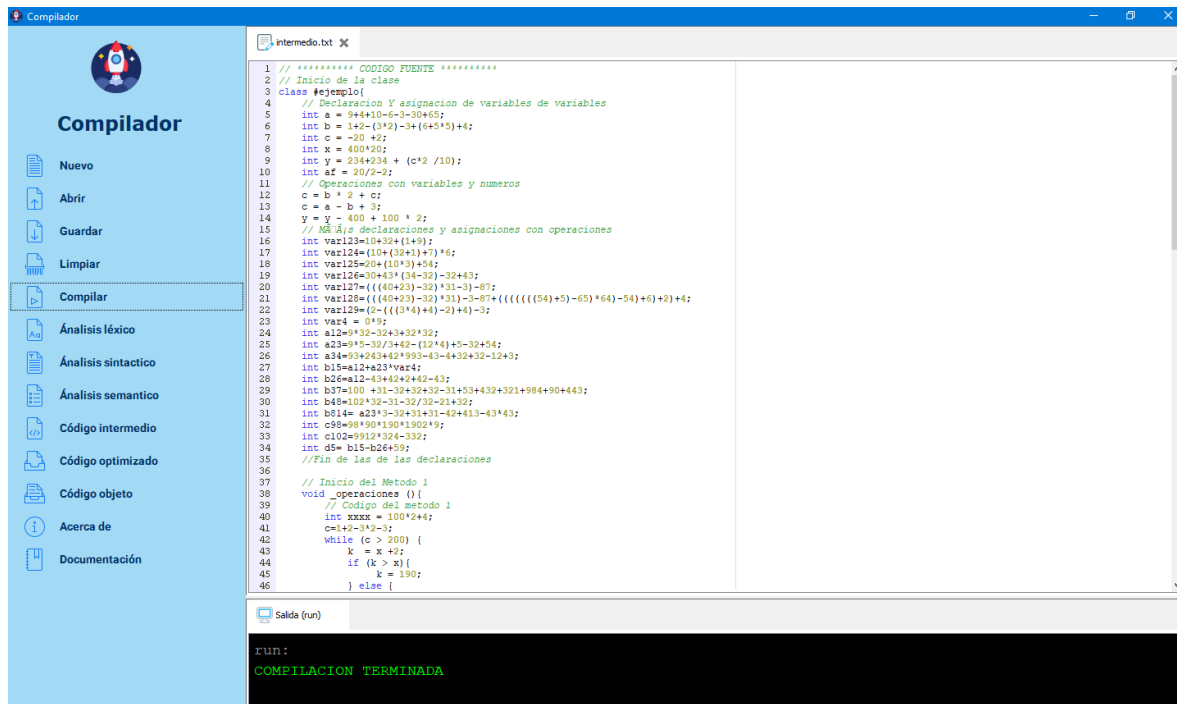
En la Figura 7, el resultado del análisis semántico.

Figura 7 Resultado del análisis semántico

Capítulo V. Analisis intermedio

En la *Figura 8* se muestra el entorno de desarrollo del compilador Coffe-Code, en la cual se tiene lo siguiente:

En el resultado de la generación del código intermedio se muestra la expresión que se evalúa, los cuádruplos que se realizan para resolver la operación y la variable con el resultado que se obtiene.



```

1 // ***** CODIGO FUENTE *****
2 // Inicio de la clase
3 class ejemplo{
4 // Declaracion Y asignacion de variables de variables
5 int a = 944+10-6-3-30+65;
6 int b = 149-(3*2)-3+(6+5*5)+4;
7 int c = -20 *2;
8 int x = 400*20;
9 int y = 234+234 + (c*2 /10);
10 int af = 20/2-2;
11 // Operaciones con variables y numeros
12 c = b * 2 + c;
13 c = a - b + 2;
14 y = y - 400 + 100 * 2;
15 // Más declaraciones y asignaciones con operaciones
16 int var123=10+32+(1+9);
17 int var124=(10+(30+1)*7)*6;
18 int var125=20+(10*3)+54;
19 int var126=30+43*(34-32)-32+43;
20 int var127=((40+23)-32)*31-3)-87;
21 int var128=((40+23)-32)*31)-3-87+((((54)+5)-65)*64)-54)+6)+2)+4;
22 int var129=(2-(((3*4)+4)-2)+4)-3;
23 int var24 = 0*9;
24 int a12=932-32+3432*32;
25 int a23=9*5-32/3+42-(12*4)+5-32+54;
26 int a34=93+243+42*993-43-4+32+32-12+3;
27 int b15=a12+23*var4;
28 int b26=a12-43+42+42+42-43;
29 int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
30 int b48=102*32-31-32/32-21+32;
31 int b614= a23*5-32+31+31-42+413-43*43;
32 int c98=98*90*190*1902*9;
33 int c102=9912*324-332;
34 int d5= b15-b26+59;
35 //Fin de las de las declaraciones
36
37 // Inicio del Metodo 1
38 void operaciones (){
39 //Codigo del metodo 1
40 int xxxxx = 100*2+4;
41 c=1+7-3*2-3;
42 while (c > 200) {
43 k = x +2;
44 if (k > x){
45 k = 190;
46 } else {

```

Figura 8 Código fuente para el análisis intermedio

En la *Figura 9*, el lado derecho se encuentra el resultado del análisis sintáctico, semántico y la generación del código intermedio.

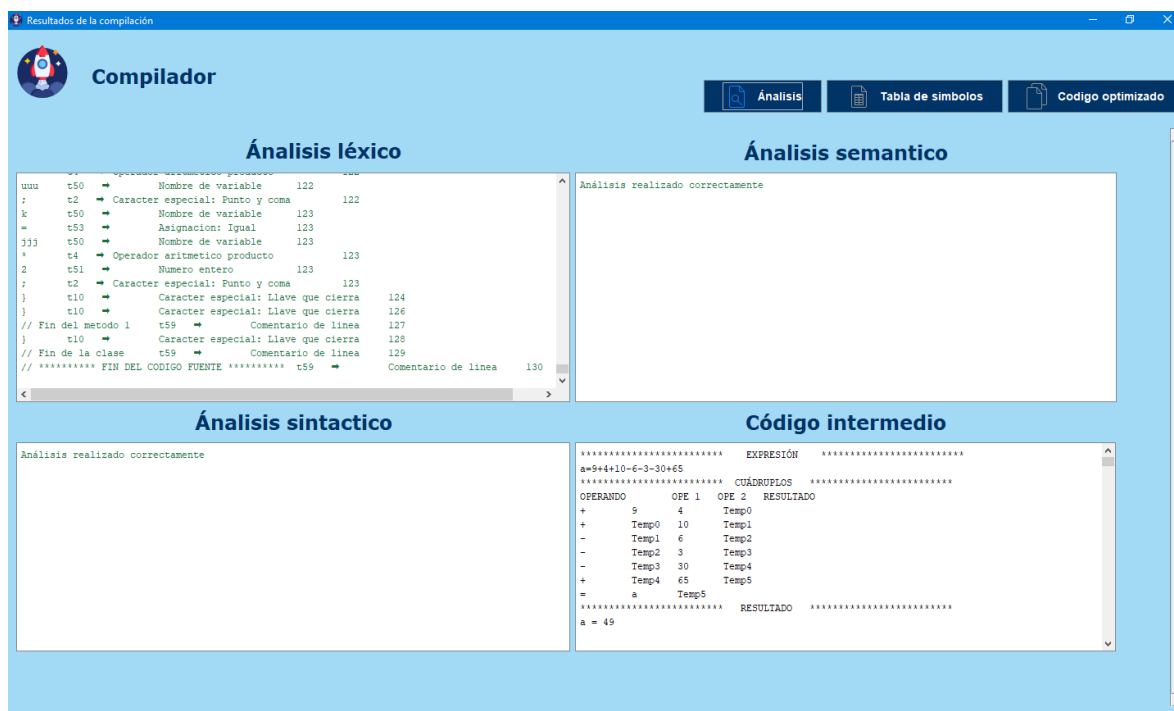


Figura 9 Resultado de la generación

Ejemplos de resultados

Ejemplo 1

Tenemos la expresión con paréntesis:

```
int b = 1 + 2 - ( 3 * 2 ) - 3 + ( 6 + 5 * 5 ) + 4;
```

Primero se han de quitar los paréntesis, para obtener una expresión más simple:

```
int b = 1 + 2 - ( 3 * 2 ) - 3 + ( 6 + 5 * 5 ) + 4;
```

Diagram showing the simplification of the expression by removing parentheses and calculating intermediate results:

$(3 * 2) = 6$

$(5 * 5) = 25$

$(6 + 25) = 31$

Posteriormente resolver la expresión, se deben aplicar la jerarquía de operadores, y las reglas de prioridad:

```
int b = 1 + 2 - 6 - 3 + 31 + 4;
```

Diagram showing the step-by-step evaluation of the expression:

$1 + 2 = 3$

$3 - 6 = -3$

$-3 - 3 = -6$

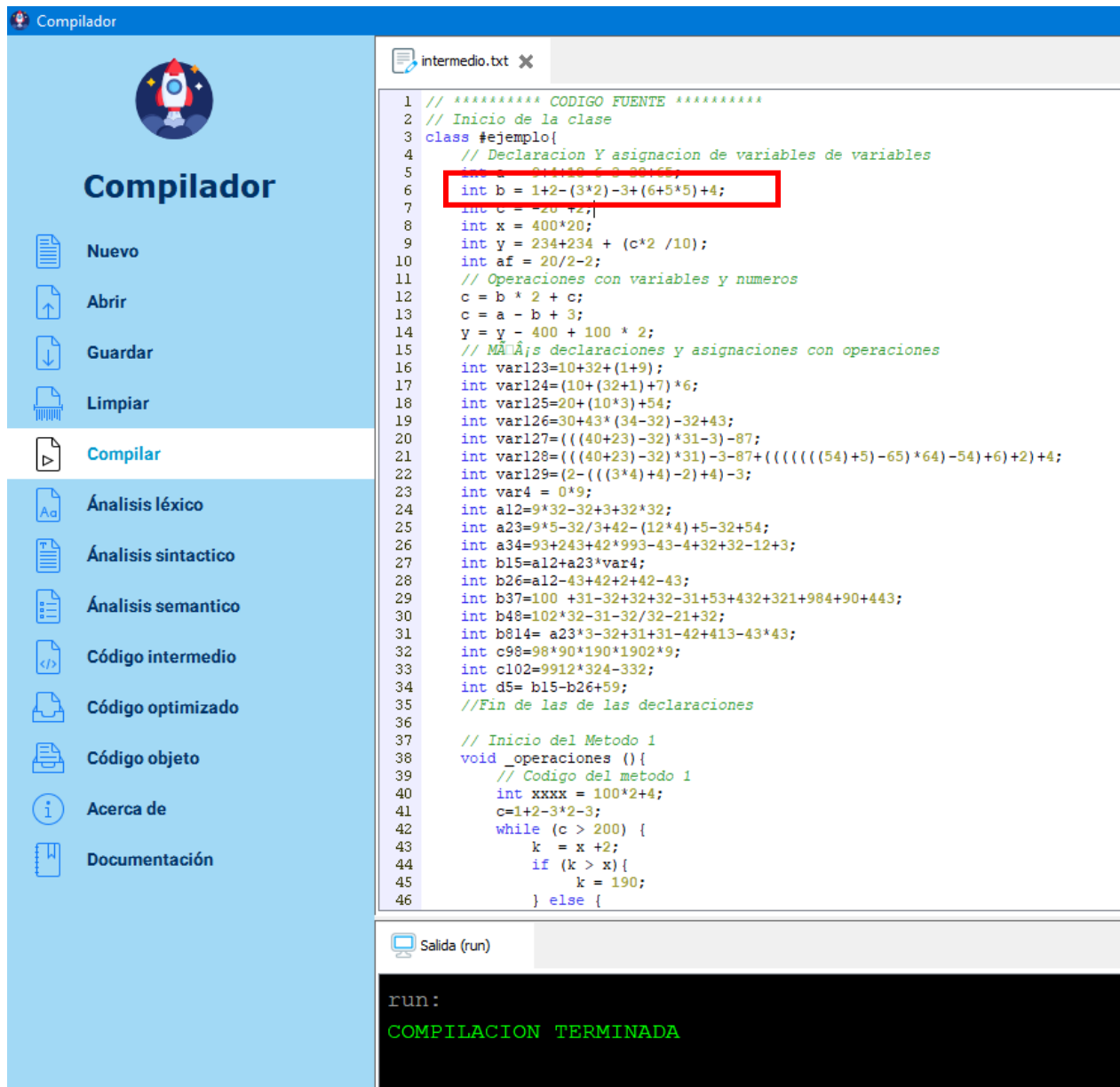
$-6 + 31 = 25$

$25 + 4 = 29$

Y finalmente asignar el resultado a la variable:

```
int b = 29
```

En la *Figura 12* se tiene el resultado de la generación del código intermedio de una operación que contiene una operación únicamente números y contiene paréntesis.



The screenshot shows a compiler application window titled "Compilador". On the left is a sidebar with icons and labels for various actions: "Nuevo", "Abrir", "Guardar", "Limpiar", "Compilar", "Análisis léxico", "Análisis sintactico", "Análisis semantico", "Código intermedio", "Código optimizado", "Código objeto", "Acerca de", and "Documentación". The main area displays a file named "intermedio.txt" containing two parts of code. The first part is the source code (CODIGO FUENTE) for a class named "ejemplo", which includes variable declarations and assignments, with line 6 highlighted in a red box: `int b = 1+2-(3*2)-3+(6+5*5)+4;`. The second part is the generated intermediate code (CÓDIGO INTERMEDIO), which consists of a series of integer assignments and arithmetic operations. At the bottom, a "Salida (run)" section shows the output: "run: COMPILACION TERMINADA".

```

1 // ***** CODIGO FUENTE *****
2 // Inicio de la clase
3 class #ejemplo{
4     // Declaracion Y asignacion de variables de variables
5     int a = 1+1+3-5+5-22+65;
6     int b = 1+2-(3*2)-3+(6+5*5)+4;
7     int c = -20+2;
8     int x = 400*20;
9     int y = 234+234 + (c*2 /10);
10    int af = 20/2-2;
11    // Operaciones con variables y numeros
12    c = b * 2 + c;
13    c = a - b + 3;
14    y = y - 400 + 100 * 2;
15    // MÃNÃs declaraciones y asignaciones con operaciones
16    int var123=10+32+(1+9);
17    int var124=(10+(32+1)+7)*6;
18    int var125=20+(10*3)+54;
19    int var126=30+43*(34-32)-32+43;
20    int var127=((40+23)-32)*31-3)-87;
21    int var128=((40+23)-32)*31)-3-87+((((((54)+5)-65)*64)-54)+6)+2)+4;
22    int var129=(2-(((3*4)+4)-2)+4)-3;
23    int var4 = 0*9;
24    int a12=9*32-32+3+32*32;
25    int a23=9*5-32/3+42-(12*4)+5-32+54;
26    int a34=93+243+42*993-43-4+32+32-12+3;
27    int b15=a12+a23*var4;
28    int b26=a12-43+42+2+42-43;
29    int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
30    int b48=102*32-31-32/32-21+32;
31    int b814= a23*3-32+31+31-42+413-43*43;
32    int c98=98*90*190*1902*9;
33    int c102=9912*324-332;
34    int d5= b15-b26+59;
35    //Fin de las de las declaraciones
36
37    // Inicio del Metodo 1
38    void _operaciones (){
39        //Codigo del metodo 1
40        int xxxx = 100*2+4;
41        c=1+2-3*2-3;
42        while (c > 200) {
43            k = x +2;
44            if (k > x){
45                k = 190;
46            } else {

```

run:
COMPILACION TERMINADA

Figura 20 Código fuente de la generación intermedio, ejemplo 1

| Código intermedio | | | |
|-------------------------|--------|--------|-----------|
| ***** EXPRESIÓN ***** | | | |
| b=1+2-(3*2)-3+(6+5*5)+4 | | | |
| ***** CUÁDRUPLOS ***** | | | |
| OPERANDO | OPE 1 | OPE 2 | RESULTADO |
| + | 1 | 2 | Temp7 |
| * | 3 | 2 | Temp8 |
| - | Temp7 | Temp8 | Temp9 |
| - | Temp9 | 3 | Temp10 |
| * | 5 | 5 | Temp11 |
| + | 6 | Temp11 | Temp12 |
| + | Temp10 | Temp12 | Temp13 |
| + | Temp13 | 4 | Temp14 |
| = | b | Temp14 | |
| ***** RESULTADO ***** | | | |
| b = 29 | | | |

Figura 31 Ejemplo 1

Ejemplo 2

Para este ejemplo, solo se usará una expresión simple de números:

int c = 20 + 2 ;

Y cuyo resultado es:

int c = 22 ;

En la *Figura 13* se observa el ejemplo de la operación que contiene únicamente números.

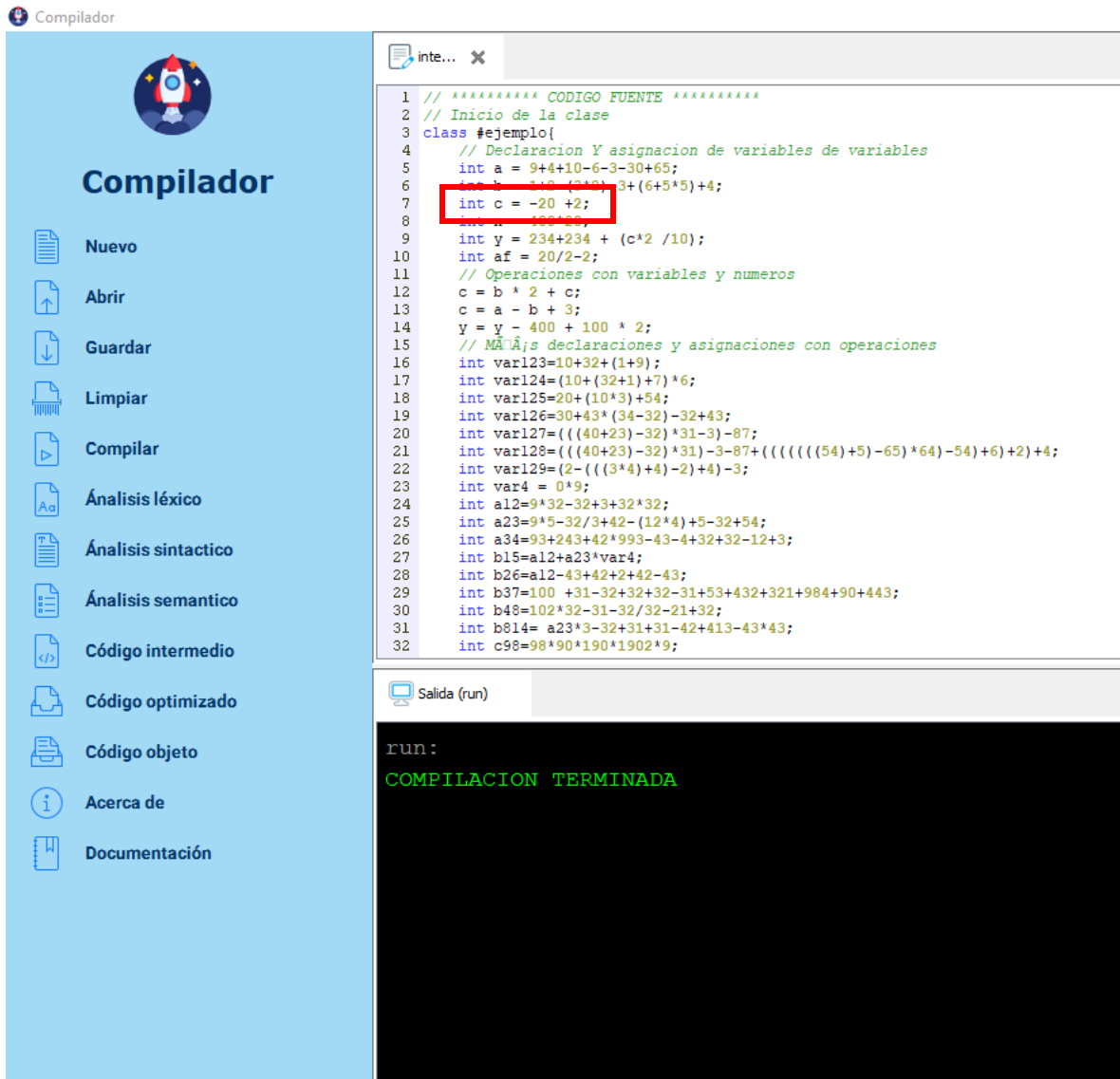


Figura 42 Código fuente de la generación intermedio, ejemplo 2

Código intermedio

```

=      b      Templ4
***** RESULTADO *****

b = 29

***** EXPRESIÓN *****

c=-20+2
***** CUÁDRUPLOS *****
OPERANDO   OPE 1   OPE 2   RESULTADO
+         -20     2      Templ6
=         c      Templ6
***** RESULTADO *****

c = -18
  
```

Figura 15 Ejemplo 2

Ejemplo 3

Cuando se trata de una expresión que contiene números y variables, estas ultimas se han de consultar desde la tabla de símbolos generada. Por ejemplo:

$$c = b * 2 + c;$$

Se buscará en la tabla el valor de b, y de c;

$$b = 29$$

$$c = 22$$

Entonces, estas variables se reemplazarán por su valor, quedando así la expresión:

$$c = 29 * 2 + 22;$$

Y al resolver la operación quedaría:

$$c = 80;$$

En la *Figura 14* se muestra una operación con un número y 2 variables, a las cuales se les asignó un valor previamente.

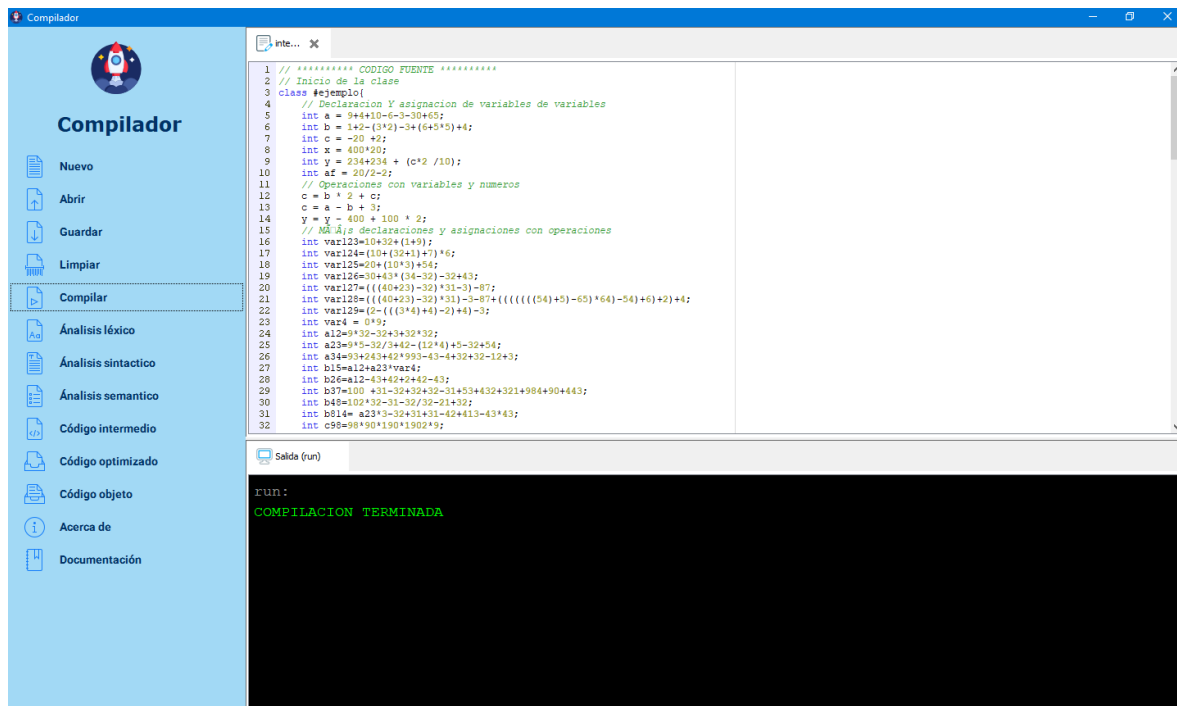


Figura 64 Código fuente de la generación intermedio, ejemplo 3

Código intermedio

```

***** EXPRESIÓN *****
c=b*2+c
***** CUÁDRUPLOS *****
OPERANDO      OPE 1  OPE 2  RESULTADO
*             b      2      Temp28
+             Temp28 c      Temp29
=             c      Temp29
***** RESULTADO *****
c = 40

***** EXPRESIÓN *****
c=a-b+3
***** CUÁDRUPLOS *****
OPERANDO      OPE 1  OPE 2  RESULTADO

```

Figura 15 Ejemplo 3

Código explicado

Para generar el código intermedio en las operaciones, principalmente se consideró que una vez sea validado el análisis semántico, posteriormente se realice la generación del código intermedio, sin embargo, el análisis semántico se realice de nuevo. Esto es para que, al momento de realizar los cuádruplos, las operaciones que llevan variables tomen el ultimo valor actualizado de la tabla de símbolos. En la Figura 16, se muestra una parte del código del análisis semántico, sin embargo, es aquí en donde se empieza a generar el código intermedio, con la variable valor de tipo String, se almacenará el valor de la operación resultante, por lo que primero hay que recorrer y agregar las operaciones en un ArrayList para después enviarla al método `conversionArrayCola`, que retornará el valor de la operación. El método `cambiaValor`, actualiza el valor de la operación de la variable en la tabla de símbolos, y posteriormente realiza el proceso de generación de cuádruplos con el método `recibeTokens`, que recibe el ArrayList, en ese mismo método es en donde se concatena la impresión de la expresión y cuádruplos, para que al final se muestre se concatene el resultado.

Finalmente se muestra la concatenación `operaciones` en el `textArea Main.objeto`.

```
String valor = Semantico.conversionArrayCola(arr); //RETORNA EL VALOR DE LA OPERACION
String tipoDatoValor = Semantico.tipoDato(valor); //RETORNA EL TIPO DE DATO DEL VALOR
if (Semantico.asignacion(tipoDatoID, tipoDatoValor)) { //VALIDA EL TIPO DE DATO DEL IDENTIFICADOR CON EL DEL VALOR
    cambiarValor(tablaSimbolos, var, valor);
    cambiarInicializada(tablaSimbolos, var, true);
    Intermedio.recibeTokens(arrInt, null);
    Intermedio.operaciones += "***** RESULTADO *****\n";
    Intermedio.operaciones += variable + " = " + valor + "\n\n";
    Main.objeto.setText(Intermedio.operaciones);
}
```

Figura 16 Impresión de la generación del código Intermedio

En la *Figura 17*, se puede apreciar cuando se agregan los tokens al ArrayList, que si llega una variable en el ArrayList no se le agrega el lexema, sino que se va a buscar a la tabla de símbolos y retorna el valor de la variable.

```

while(agrega){
    if (tokenSiguiente.getNumToken() == VARIABLE) {
        // Validar si esta declarada
        if (!isDuplicated(tablaSimbolos, tokenSiguiente.getLexema())) {
            errorNoDeclarada(tokenSiguiente);
            hacer = false;
        } else {
            if (valor(tablaSimbolos, tokenSiguiente.lexema) == null) {
                errorNoIniciada(tokenSiguiente);
                hacer = false;
            } else {
                Lexema a3 = new Lexema();
                a3.setLexema(tokenSiguiente.lexema);
                a3.setTipoToken(valor(tablaSimbolos, tokenSiguiente.lexema));
                arrInt.add(a3);
                arr.add(valor(tablaSimbolos, tokenSiguiente.lexema));
            }
        }
    }
}
} else {

```

Figura 17 Llenar el ArrayList de tokens

En la *Figura 18*, se convierte el ArrayList de tokens, en una cola de operaciones, que después es ingresada al método posfijo que retornará la misma cola, pero con la notación posfija, después, esta cola con notación Posfija se ingresará en el método operaciones que retornará el valor de la operación. Este valor es el que es retornado.

```

public static String conversionArrayCola(ArrayList<String> arr) {
    String valor;
    if (arr.size() == 1) {
        return arr.get(0);
    } else {
        ColaD colaOperaciones = new ColaD();
        for (int i = 0; i < arr.size(); i++) {
            Nodo n = new Nodo(arr.get(i), -1);
            colaOperaciones.inserta(n, null);
        }
        Semantico s = new Semantico();
        ColaD notacionPostfija = s.posfijo(colaOperaciones);

        valor = String.valueOf(s.operaciones(notacionPostfija));
    }

    return valor;
}

```

Figura 18 Método conversionArrayCola()

En la *Figura 19*, se muestra el código que se implementó para realizar la generación de código intermedio. En el método recibeTokens, recibe el ArrayList de Tokens que contienen la expresión matemática del código fuente para pasarlos a una cola y después procesarlos. Como se puede ver, se recorre el ArrayList para imprimir la expresión que será transformada, después se prepara la impresión de cómo se van a mostrar los cuádruplos en el compilador y después se llena una cola de operaciones. Luego esta es ingresada al método posfijo, método regresará una cola

ordenada. Posteriormente, la cola notación Posfija será ingresada a operacionesIntermedio; método que generará cuádruplos.

```
public static void recibeTokens(ArrayList<Lexema> lx, ArrayList<TablaSimbolos> tSimbolos) {

    String tr = "***** EXPRESIÓN *****\n";
    for (int i = 0; i < lx.size(); i++) {
        tr += lx.get(i).getLexema();
    }
    tr += "\n***** CUÁDRUPLOS *****\n";
    tr += "OPERANDO\toPE 1\toPE 2\tRESULTADO\n";
    operaciones += tr;

    ColaD colaOperaciones = new ColaD();
    for (int i = 0; i < lx.size(); i++) {
        Nodo n = new Nodo(lx.get(i).getLexema(), -1);
        colaOperaciones.inserta(n, null);
    }
    ColaD notacionPostfija = Intermedio.posfijo(colaOperaciones);
    Intermedio.operacionesIntermedio(notacionPostfija);
}
```

Figura 19 Método recibeTokens()

Para generar los cuádruplos se ocupó del algoritmo de notación posfija.

En la *Figura 20*, se muestra el método posfijo, al que recibe una cola dinámica, que contendrá por cada nodo un lexema que puede ser, un operador o un operando, y retorna una cola con los lexemas ordenados para que pueda realizarse posteriormente la operación. Se declara una cola dinámica llamada pResultado; esta contendrá los lexemas finales, una Pila dinámica; que contendrá los operadores y los paréntesis.

El primer while permite que se recorra la cola recibida, hasta que se vacíe. En la variable s, de tipo String, se almacena el lexema que es eliminado de la cola, luego se manda a comparar para saber si se trata de un paréntesis que abre, cierra, o cualquier operador, si retorna que es un operador, ha de preguntar si la pila está vacía, si así lo es, simplemente se introduce a la pila de operadores, sino se ha de evaluar con otro operador y comparar cual es de mayor prioridad, si s2 que es el operador que estaba al tope de la pila, tiene mayor o igual prioridad, entonces en la cola resultado se inserta s2, y s, se manda a un método que determinará su movimiento, si s2 es menor que s, entonces se ingresan a la pila de operadores ambas variables. La prioridad es determinada por el método prioridad(variable) cuyo retorno es un número.


```

public ColaD posfijo(ColaD cola) {
    ColaD pResultado = new ColaD();
    PilaD pOperadores = new PilaD();
    while (cola.getF() != null) {
        String s = cola.elimina(null).getS();
        if (isOperadorS(s)) {
            if (isOperador(s)) {
                if (pOperadores.getTope() != null) {
                    String s2 = pOperadores.elimina(null).getS();
                    if (isOperador(s2)) {
                        if (prioridad(s2) >= prioridad(s)) {
                            pResultado.inserta(new Nodo(s2, -1), null);
                            insertaPOperadores(s, pOperadores, pResultado);
                        } else if (prioridad(s2) < prioridad(s)) {
                            pOperadores.inserta(new Nodo(s2, -1), null);
                            pOperadores.inserta(new Nodo(s, -1), null);
                        } else {
                            pOperadores.inserta(new Nodo(s2, -1), null);
                            pOperadores.inserta(new Nodo(s, -1), null);
                        }
                    } else {
                        pOperadores.inserta(new Nodo(s2, -1), null);
                        pOperadores.inserta(new Nodo(s, -1), null);
                    }
                } else {
                    pOperadores.inserta(new Nodo(s, -1), null);
                }
            } else {
                pOperadores.inserta(new Nodo(s, -1), null);
            }
        }
    }
}

```

Figura 20 Método posfijo(), parte 1

En la *Figura 21*, que es la segunda parte del método posfijo, muestra que si *s* es un paréntesis que abre, entonces se ingresa a la pila de operadores hasta que llegue su otro paréntesis que cierra, si *s* es ese paréntesis que cierra, entonces vaciara la pila hasta que se encuentre un su paréntesis que abre y cada nodo que se elimine se insertara en la cola resultado, de esta manera se eliminan los paréntesis de la expresión. Y finalmente en el while que este subrayado, dice que, si la pila de operadores sigue llena, entonces se vaciará la pila a la cola resultado.

```

    }
    } else if (s.equals("(")) {
        pOperadores.inserta(new Nodo(s, -1), null);
    } else if (s.equals("(")) {
        while (pOperadores.getTope() != null) {
            if (pOperadores.getTope().getS().equals("(")) {
                pOperadores.elimina(null);
                break;
            } else {
                String op = pOperadores.elimina(null).getS();
                pResultado.inserta(new Nodo(op, -1), null);
            }
        }
    }
    } else {
        pResultado.inserta(new Nodo(s, -1), null);
    }
}

while (pOperadores.getTope() != null) {
    String op = pOperadores.elimina(null).getS();
    pResultado.inserta(new Nodo(op, -1), null);
}

return pResultado;
}

```

Figura 21 Método posfijo() , parte 2

En la *Figura 22*, se muestra el método insertaPOperadores, este método es el que se llama cuando la prioridad de s2 es mayor a s, y s necesita saber si entra en la pila de operadores o en la cola resultado, por lo que se ha de eliminar otro elemento de la pila operadores para decidir.

```

public void insertaPOperadores(String s, PilaD pOperadores, ColaD pResultado) {
    if (pOperadores.getTope() != null) {
        String s2 = pOperadores.elimina(null).getS();
        if (isOperador(s2)) {
            if (prioridad(s2) >= prioridad(s)) {
                pOperadores.inserta(new Nodo(s, -1), null);
                pResultado.inserta(new Nodo(s2, -1), null);
            } else if (prioridad(s) > prioridad(s2)) {
                pOperadores.inserta(new Nodo(s2, -1), null);
                pOperadores.inserta(new Nodo(s, -1), null);
            } else {

```

Figura 22 Método insertaPOperadores()

En la *Figura 23*, se muestra el método prioridad, que recibe el operador en String, y retorna un número según su prioridad de operación. Como se puede apreciar, el ^ es el que tiene mayor prioridad ya que primero se han de hacer las potencias, después y con la misma jerarquía las divisiones "/", multiplicaciones "*", y módulos "%", después las sumas "+" y restas "-", en caso de las operaciones matemáticas, después los operadores relacionales o de comparación y finalmente los operadores lógicos, si no es ninguno de ellos, por defecto se ha de retornar 1;

```
private int prioridad(String operador) {
    switch (operador) {
        case "^":
            return 6;
        case "/":
        case "*":
        case "%":
            return 5;
        case "+":
        case "-":
            return 4;
        case ">":
        case "<":
        case ">=":
        case "<=":
        case "==":
        case "!=":
            return 3;
        case "&&":
        case "||":
        case "!":
            return 2;
        default:
            break;
    }
    return 1;
}
```

Figura 23 Método prioridad()

En la Figura 24, se determina si es un operador matemático, relacional, o lógico, o si es un paréntesis que abre o cierra.

```
public boolean isOperadorS(String s) {
    return s.equals("%") || s.equals("*") || s.equals("^") ||
        s.equals("/") || s.equals("+") || s.equals("-") || s.equals("!") ||
        s.equals("&&") || s.equals("||") || s.equals("<") || s.equals(">") ||
        s.equals(">=") || s.equals("<=") || s.equals("==") || s.equals("!=") ||
        s.equals("(") || s.equals(")");
}
```

Figura 24 Método isOperadorS()

Ahora que la cola Resultado se ha acomodado sin paréntesis y respetando la prioridad, se ha de evaluar, para mostrar el resultado de la operación, para ello en el método operaciones que se muestra en la Figura 24. Este método utiliza una pila que contendrá operandos, y utilizara la cola resultada que se obtuvo anteriormente. El primer while permite recorrer la cola hasta que se vacíe, en la variable resultado de tipo Object, se almacenará el resultado tanto parcial como final. Y en res[] es la variable que contendrá el resultado por cada operación, entonces, se elimina un

nodo de la cola resultado y se almacena en la variable s, se compara si s es un operador, si no lo es, entonces se insertará en la pila de operandos, si es un operador entonces se eliminarán dos nodos de la pila de operandos, se le determinan de que tipos de datos son (Cadena, Entero, flotante o booleano), y se envía como parámetros al método `expresionFinal()` que retornará el resultado en la posición del arreglo 1, este resultado se introduce de nuevo a la pila de operandos, y se actualiza el valor de resultado por el `res[1]`.

```
public static Object operaciones(ColaD pResultado) {
    PilaD pOperacion = new PilaD();
    Object resultado = 0;
    String res[];
    while (pResultado.getF() != null) {
        String s = pResultado.elimina(null).getS();
        if (isOperador(s)) {
            String op2 = pOperacion.elimina(null).getS();
            String op1 = pOperacion.elimina(null).getS();

            String tipoOp1 = tipoDato(op1);
            String tipoOp2 = tipoDato(op2);

            res = expresionFinal(op1, op2, s, tipoOp1, tipoOp2);
            pOperacion.inserta(new Nodo(String.valueOf(res[1]), -1), null);
            resultado = res[1];
        } else {
            pOperacion.inserta(new Nodo(s, -1), null);
        }
    }
    return resultado;
}
```

Figura 25 Método operaciones ()

Al mismo tiempo que se ejecuta el método `operaciones`, se realiza el método `operacionesIntermedio`, que es el método que almacenará en los cuádruplos. Este método recibe la cola `Resultado`, y recorrerá cada posición hasta que se vacíe la cola, se eliminará un nodo de la cola y se almacenará en la variable s, después se determinará si es un operador, si no lo es, entonces se agregará a la pila de operandos, si lo es, entonces creará un cuádruplo, s será el operador, se eliminarán dos nodos de la pila de operandos, el primero será el operando 1, y el segundo el operando 2, después se chequea si el operando es un "=", si lo es, entonces ya no se creará una temporal, si no lo es, entonces se creará una temporal que se incrementa en uno cada vez que se cree un cuádruplo. Esta temporal se almacenará en la pila de operandos ya que representa el resultado que se obtiene de la operación.

Luego se concatena en la forma prefija: el operador, el operando 1, el operando 2, la temporal.

```

public static Object operacionesIntermedio(ColaD pResultado) {
    PilaD pOperacion = new PilaD();
    Object resultado = 0;
    while (pResultado.getF() != null) {
        String s = pResultado.elimina(null).getS();
        Cuaduplos cI = new Cuaduplos();
        if (isOperador(s)) {
            String op2 = pOperacion.elimina(null).getS();
            String op1 = pOperacion.elimina(null).getS();
            cI.setOp(s);
            cI.setOp1(op1);
            cI.setOp2(op2);
            if (s.equals("=")) {
                cI.setTemp(" ");
            } else {
                cI.setTemp("Temp" + temp);
            }
            pOperacion.inserta(new Nodo("Temp" + temp, -1), null);
            temp++;
            operaciones += cI.getOp() + "\t " + cI.getOp1() + "\t " + cI.getOp2() + "\t " + cI.getTemp() + "\n";
        } else {
            pOperacion.inserta(new Nodo(s, -1), null);
        }
    }
    return resultado;
}

```

Figura 26 Método operacionesIntermedio()

En la Figura 25, se muestra el código que hace la operación. Como se puede ver, recibe el operando 1, el operando 2, el operador, el tipo de dato del operando 1, y el del operando 2. Para que la operación sea correspondida, se verifica que ambos tipos de datos sean compatibles, en este caso si ambos son INTEGER, entonces ingresará a otro case para checar de que operador se trata, en este caso es una suma, y se procede a parsear a los operandos, según el tipo de dato. Después se realiza la operación normal y se almacena en la variable resultados en la posición 1, y esta es la que se retorna. Si por alguna razón no se puede parsear, se mandará un False en la variable.

```

private static String[] expresionFinal(String op1, String op2, String operador, String tipoDato1, String tipoDato2) {
    String resultados[] = new String[3];

    if (tipoDato1.equals("INTEGER") && tipoDato2.equals("INTEGER")) {
        switch (operador) {
            case "+":
                try {
                    int var1 = Integer.parseInt(op1);
                    int var2 = Integer.parseInt(op2);

                    resultados[1] = (var1 + var2) + "";
                    resultados[0] = "TRUE";
                } catch (NumberFormatException e) {
                    resultados[0] = "FALSE";
                }
                break;
            case "-":

```

Figura 27 Método expresionFinal()

Capítulo VI. Código optimizado

En el resultado de la optimización se muestra en una nueva ventana la cual se muestra en la *Figura 28*, en ella se muestra el código fuente y el optimizado, al finalizar la ejecución de cada código se genera un archivo de texto.

Adicional, en la parte inferior se muestran los resultados de la ejecución, en los cuales se puede observar el tiempo de ejecución (mostrado en minutos, segundos y milisegundos) y el tamaño del archivo que se generó de ambos códigos (mostrado en bytes).

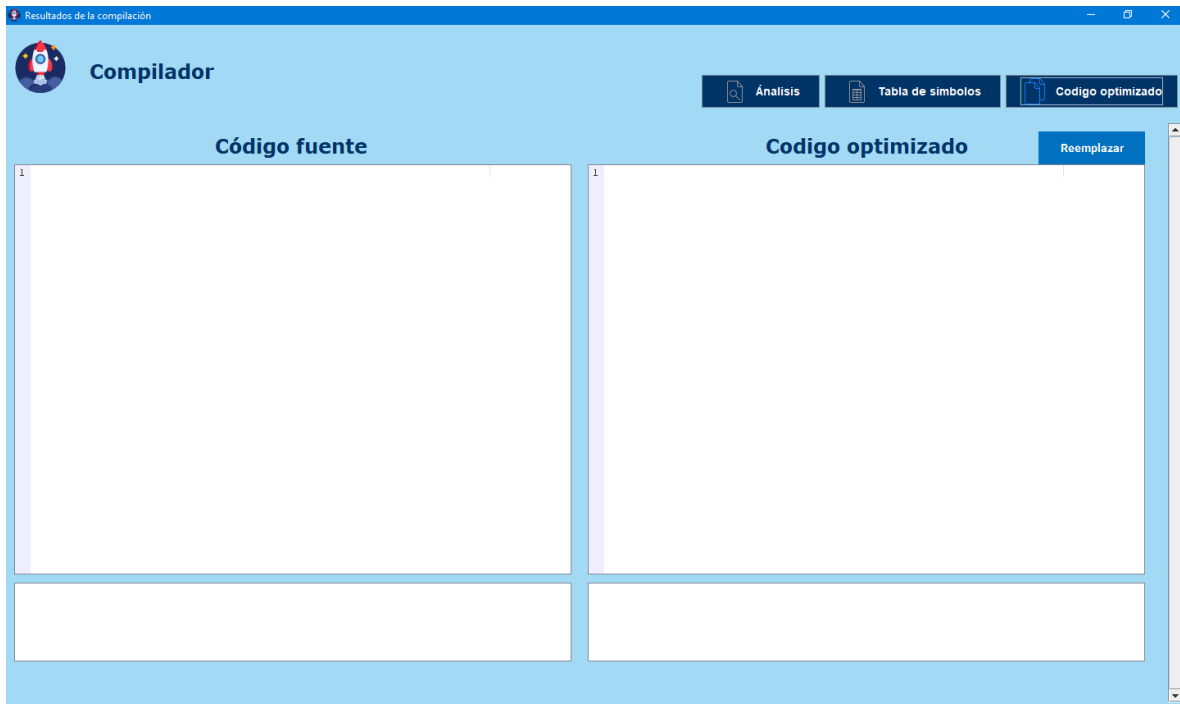


Figura 28 Pantalla de resultado del código optimizado

Ejemplo de optimización

Para el ejemplo de la optimización se usó el código de la *Figura 29*

```
// ***** CÓDIGO FUENTE *****
// Inicio de la clase
class #ejemplo{
    // Declaración Y asignación de variables de variables
    int a = 9+4+10-6-3-30+65;
    int b = 1+2-(3*2)-3+(6+5*5)+4;
    int c = -20 +2;
    int x = 400*20;
    int y = 234+234 + (c*2 /10);
    int af = 20/2-2;
    // Operaciones con variables y números
    c = b * 2 + c;
    c = a - b + 3;
    y = y - 400 + 100 * 2;
    // Más declaraciones y asignaciones con operaciones
    int var123=10+32+(1+9);
    int var124=(10+(32+1)+7)*6;
    int var125=20+(10*3)+54;
    int var126=30+43*(34-32)-32+43;
    int var127=((40+23)-32)*31-3-87;
    int var128=((40+23)-32)*31-3-87+((((54)+5)-65)*64)-54+6)+2)+4;
    int var129=(2-(((3*4)+4)-2)+4)-3;
    int var4 = 0*9;
    int a12=9*32-32+3+32*32;
    int a23=9*5-32/3+42-(12*4)+5-32+54;
    int a34=93+243+42*993-43-4+32+32-12+3;
    int b15=a12+a23*var4;
    int b26=a12-43+42+2+42-43;
    int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
    int b48=102*32-31-32/32-21+32;
    int b814= a23*3-32+31+31-42+413-43*43;
    int c98=98*90*190*1902*9;
    int c102=9912*324-332;
    int d5= b15-b26+59;
    //Fin de las de las declaraciones

    // Inicio del Método 1
    void _operaciones (){
        // Codigo del metodo 1
        int xxxx = 100*2+4;
        c=1+2-3*2-3;
        d5 = c*c;
        c102 = xxxx - c -d5;

        // Comentario ej
        y = 234+234;
        af = 20/2-2;
        x = af+67;
        int d = (132+23*(34/3)+234);
        int s = d/2;
        int ent3 = 10;
        int ent7 = 12+14*2;
        // Operaciones con variables
        x = a +b+c;
        d5 = xxxx+35+234*456-345;
        a = 35+234*456-345;
        int j= 21*3+14-20;
        // Operaciones con números
        int fg = 20*35+234*456-345;
        int u = a+20*35+234*456-345;
        a = b *2 *c;
        a = 436+235+23+20*35+234*456-345;
        b = 400+234;
        int k = 234;
        b = a*k;
        a = a*k+2000;
        k = 234+a*c;
        // Asignmaciones de un solo número
        b = -1;
        xxxx = 24;
        k = 2000;
    }
    // Fin del metodo 1
}
// Fin de la clase
// ***** FIN DEL CÓDIGO FUENTE *****
```

Ejecución del código fuente

Una vez que en el compilador se tiene el código fuente, se puede ejecutar para visualizar los resultados de los análisis.

Cuando finaliza la ejecución, se muestra en pantalla un mensaje indicando que se han generado los archivos donde se guarda el código fuente y el optimizado, tal y como se muestra en la *Figura 30*, en la cual también podemos visualizar el tamaño de estos.

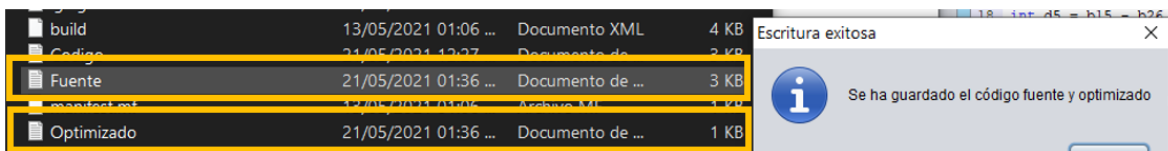


Figura 30 Archivos generados

En la *Figura 31* se muestra la pantalla principal, donde se puede observar la ejecución correcta del análisis léxico, sintáctico, semántico e intermedio.

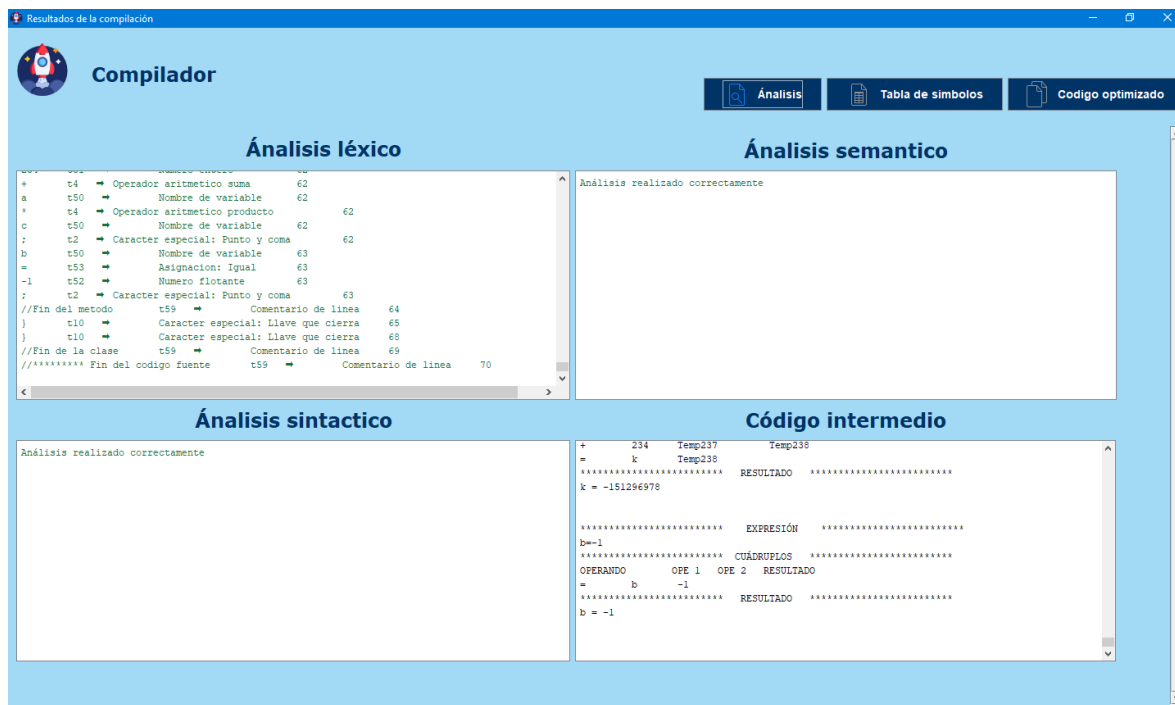


Figura 31 Ejecución del código fuente

Resultado de la optimización – fuente

Como se mencionó anteriormente, el resultado del código optimizado se muestra en una ventana adicional, la cual se muestra en la *Figura 32*.

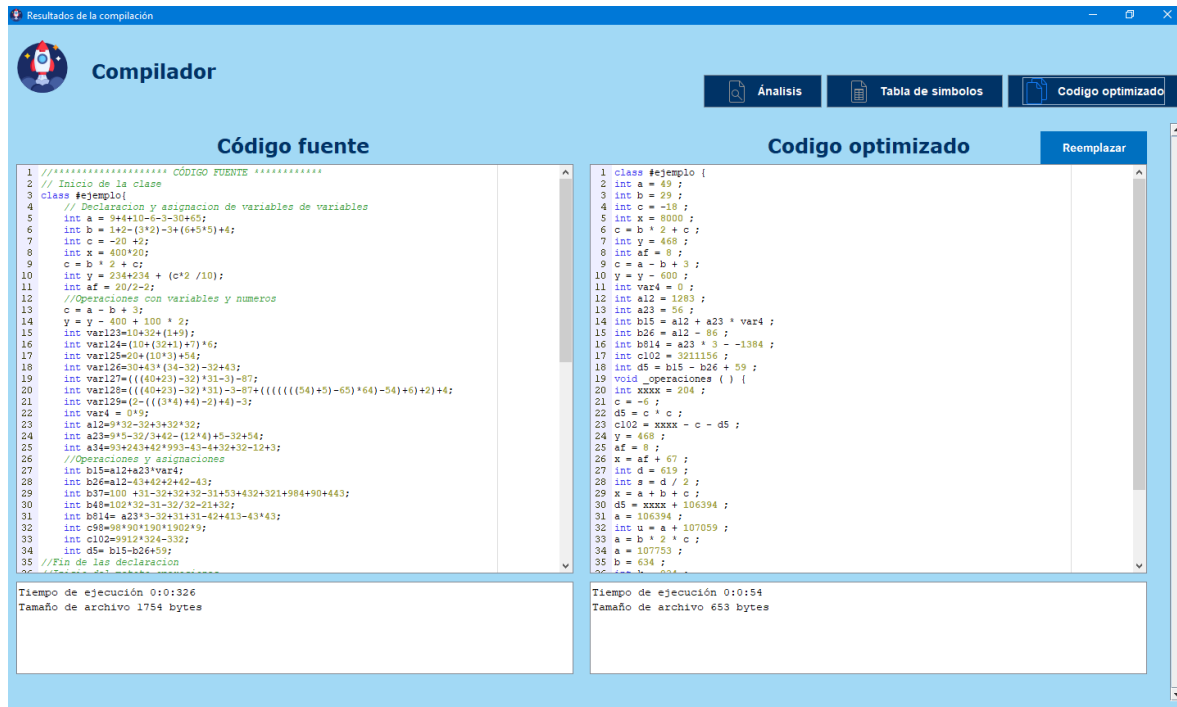


Figura 32 Resultado del código optimizado

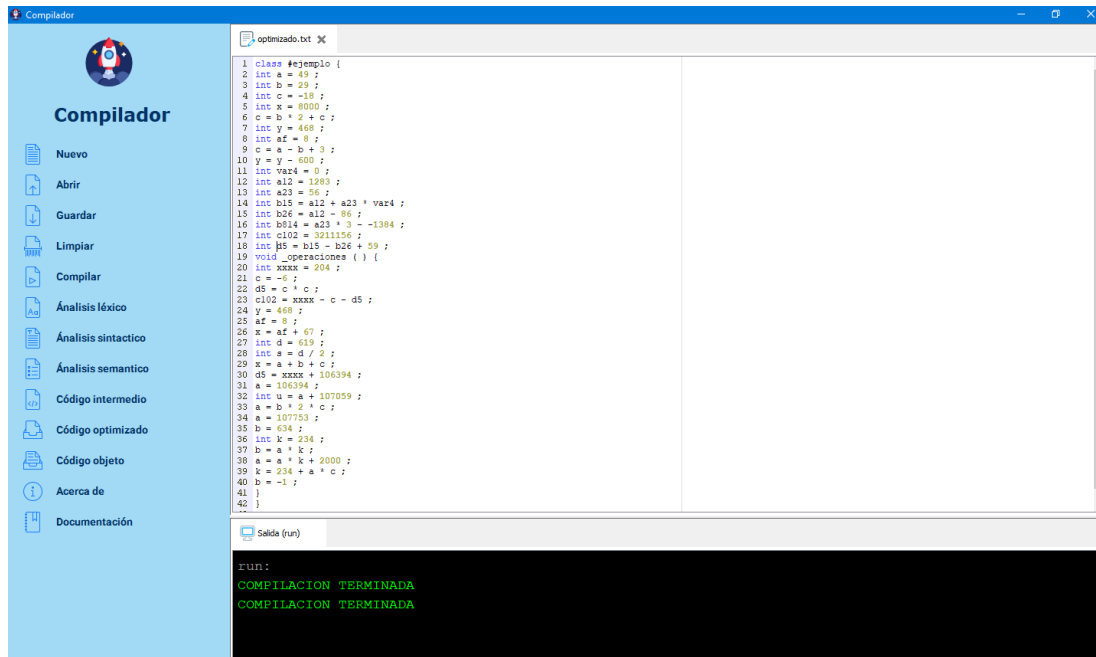
En la figura anterior se observan los resultados de la optimización y a continuación, se describirán las optimizaciones obtenidas:

- **Número de líneas:** El código fuente tiene un total de **73 líneas**, mientras que el optimizado tiene **43 líneas**.
- **Tiempo de ejecución:** El código fuente se ejecutó en **326 milisegundos** y el optimizado en **54 milisegundos**.
- **Tamaño de los archivos:** El archivo del código fuente pesa **1754 bytes** y el del optimizado **653 bytes**.

En conclusión, se obtuvo una reducción total de 30 líneas de código, 272 milisegundos y 1101 bytes.

Ejecución del código optimizado

Para mostrar la ejecución del código optimizado, se copiaron las 44 líneas obtenidas del resultado de la optimización, tal y como se muestra en la *Figura 33*, en la cual también se puede observar que el análisis léxico, sintáctico, semántico e intermedio se realizan de manera adecuada.



The screenshot shows a compiler application window titled 'Compilador'. On the left is a sidebar with icons and labels for various actions: 'Nuevo', 'Abrir', 'Guardar', 'Limpiar', 'Compilar', 'Análisis léxico', 'Análisis sintáctico', 'Análisis semántico', 'Código intermedio', 'Código optimizado', 'Código objeto', 'Acerca de', and 'Documentación'. The main area displays a file named 'optimizado.txt' containing 44 lines of optimized C code. The code includes variable declarations, arithmetic operations, and control structures. At the bottom, a 'Salida (run)' panel shows the output: 'run: COMPILACION TERMINADA' and 'COMPILACION TERMINADA'.

```

1 class #ejemplo {
2 int a = 49 ;
3 int b = 29 ;
4 int c = -18 ;
5 int x = 8000 ;
6 c = b * 2 + c ;
7 int y = 468 ;
8 int af = 8 ;
9 c = a - b + 3 ;
10 y = y - 600 ;
11 int var4 = 0 ;
12 int a12 = 1283 ;
13 int a23 = 56 ;
14 int b15 = a12 + a23 * var4 ;
15 int b26 = a12 - 86 ;
16 int b14 = a23 * 3 - -1384 ;
17 int c102 = 3211156 ;
18 int j5 = b15 - b26 + 59 ;
19 void _operaciones ( ) {
20 int xxxx = 204 ;
21 c = -6 ;
22 d5 = c * c ;
23 c102 = xxxx - c - d5 ;
24 y = 468 ;
25 af = 0 ;
26 x = af + 67 ;
27 int d = 619 ;
28 int s = d / 2 ;
29 x = a + b + c ;
30 d5 = xxxx + 106394 ;
31 s = 106394 ;
32 int u = a + 107059 ;
33 a = b * 2 * c ;
34 a = 107753 ;
35 b = 634 ;
36 int k = 234 ;
37 b = a * k ;
38 a = a * k + 2000 ;
39 k = 234 + a * c ;
40 b = -1 ;
41 }
42 }
  
```

run:
COMPILACION TERMINADA
COMPILACION TERMINADA

Figura 33 Código optimizado

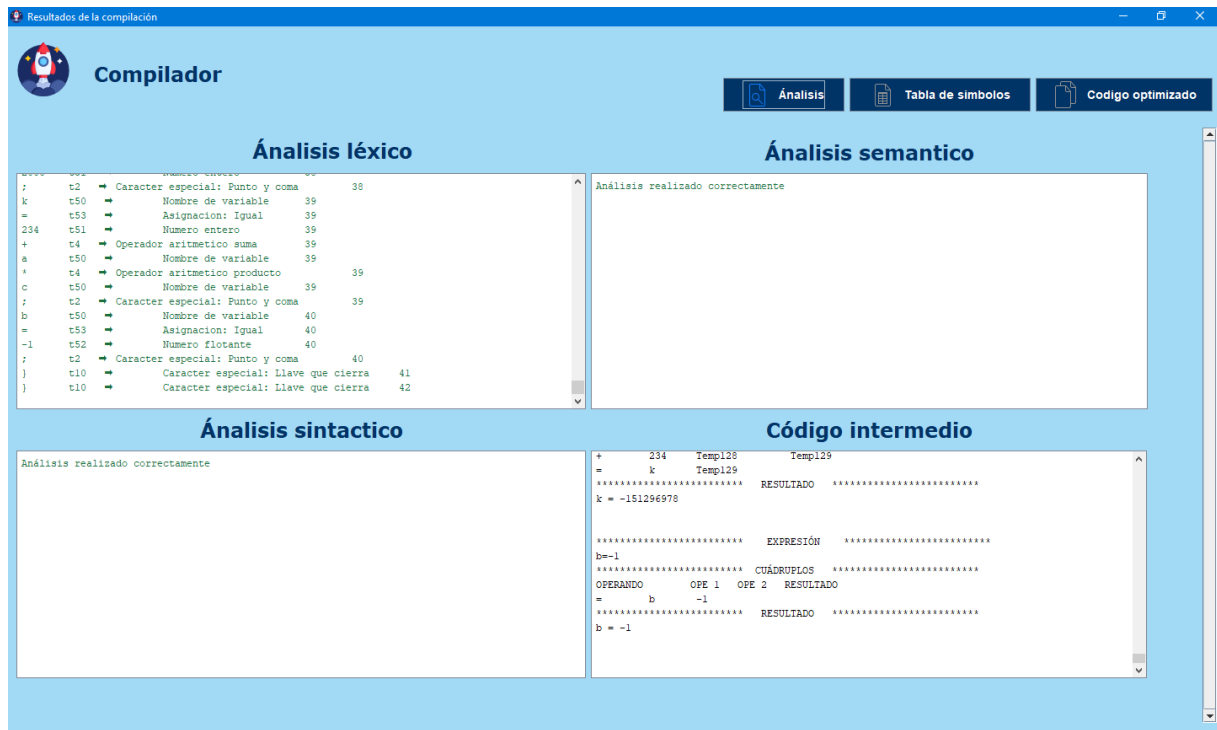


Figura 34 Ejecución del código optimizado

De igual forma, si ingresamos a la carpeta podemos observar los archivos generados y su tamaño, como en la Figura 35.




| | | | | |
|---|-------------|----------------------|------------------|------|
|  | Fuente | 21/05/2021 01:24 ... | Documento de ... | 1 KB |
|  | manifest.mf | 13/05/2021 01:06 ... | Archivo MF | 1 KB |
|  | Optimizado | 21/05/2021 01:24 ... | Documento de ... | 1 KB |

Figura 35 Archivos código optimizado

Resultados de la optimización – optimizado

En la *Figura 36* se muestra el resultado del código optimizado.

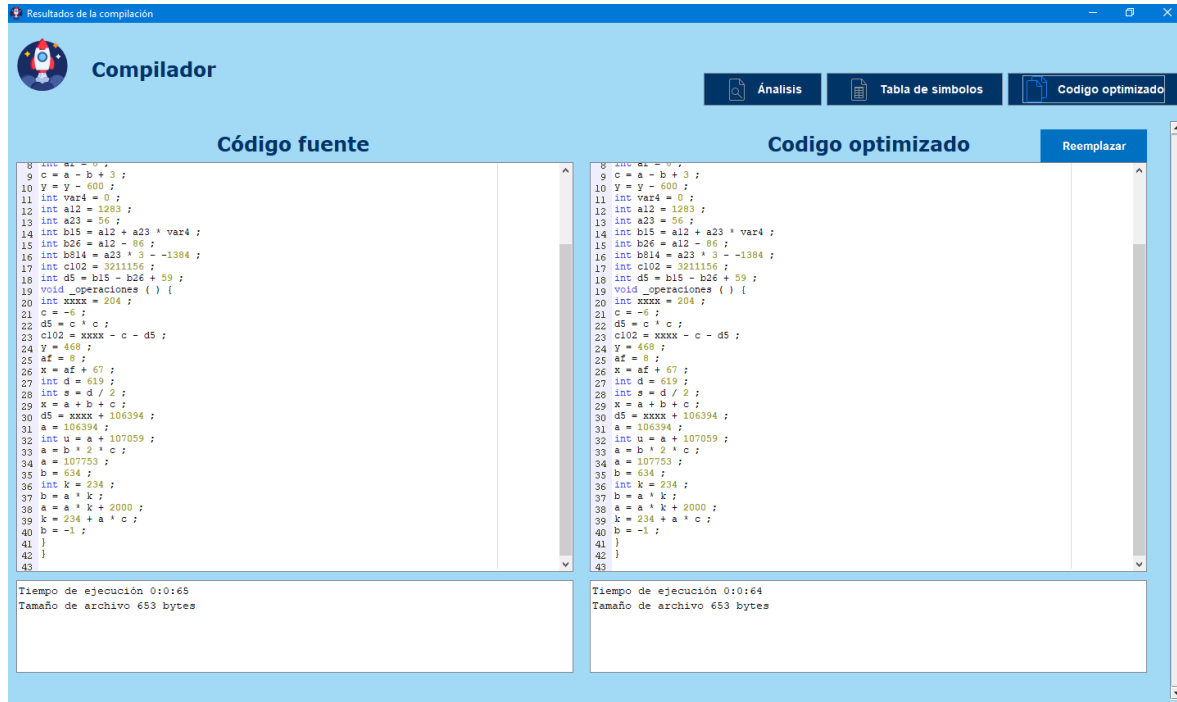


Figura 36 Resultados del código optimizado

En la figura anterior se observan los resultados de la optimización, con los cuales demostramos que una vez que una vez que el compilador genere el código optimizado a partir de un código fuente, este lo podemos colocar como nuestro nuevo código fuente y todos los análisis se ejecutaran de manera correcta.

Técnicas de Optimización

Para la optimización se emplearon las siguientes técnicas

- Eliminación de código muerto: eliminación de comentarios, tabuladores y espacios en blanco de línea.
- Reestructura de código: Primero declaración de variables, después métodos.
- Eliminación de variables declaradas sin valor.
- Eliminación de variables declaradas sin uso.
- Intercambio de asignaciones por declaraciones.
- Transformaciones algebraicas, reemplazo de operaciones costosas por otras menos costosas.
- Simplificaciones algebraicas.

Código explicado

En la Figura 37, se muestra el método que es llamado al momento de optimizar el código fuente. En este se llaman a los métodos de optimización. Por ejemplo, primero es llamado el método que hará la eliminación de comentarios para reducir el número de iteraciones del ArrayList, después pasa a una reestructura que permitirá que sea más rápido hacer cambios en el código fuente, seguido de la simplificación de operaciones costosas, seguido de eliminación de variables sin usar. Después se hace una segunda revisión de las técnicas anteriores al menos por dos veces para asegurar una optimización completa. Seguido de la sustitución de asignaciones de variables por las declaraciones sin valor. Finalmente se imprime el resultado de la lista ya reducida en un formato sin tabuladores ni espacios en blanco.

```
public void optimizacion() {  
    ArrayList<analisis.Lexema> lista2 = new ArrayList<>();  
    //Tecnica 1 quitar comentarios  
    lista2 = optimizaComentarios(lista2);  
    //Tecnica 2 Estructura  
    lista2 = optimizarEstructura(lista2);  
    //Tecnica 3 Reduccion Operaciones  
    lista2 = optimizarVariablesOp(lista2);  
    //Tecnica 4 Variables no usadas  
    lista2 = optimizaVariablesSinUso(lista2);  
    //Tecnica 5 Reduccion Operaciones Revision 2  
    lista2 = optimizarVariablesOp(lista2);  
    //Tecnica 6 Variables no usadas Revision 2  
    lista2 = optimizaVariablesSinUso(lista2);  
    //Tecnica 7 Asignaciones por declaraciones, si no hubieron cambios entr  
    lista2 = optimizaAsignacionesInnecesarias(lista2);  
    //Tecnica 8 Sustitucion de variables  
    // lista2 = optimizaOperaciones(lista2);  
    //Tecnica 8 Quitar espacios y tabuladores  
    imprimeOptimizacion(lista2);  
}
```

Figura 37 Método optimizacion()

En la Figura 38, se muestra el método que optimiza las asignaciones innecesarias por las declaraciones. Primero se recorre la lista para saber si la variable que se acaba de crear tiene un valor, si lo tiene ocupamos un indicador setRenglon y colocamos 1, si no lo está quiere decir que la variable puede declararse a partir de su primera declaración.

```
private ArrayList<Lexema> optimizaAsignacionesInnecesarias(ArrayList<Lexema> lista2) {
// si no hubieron cambios entre una asignacion y otra o no la variable no se ocupo entre una asignacion y otra
ArrayList<Lexema> arrLex = new ArrayList<>();
//Agrega variables
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getNumToken() == 1) {
        Lexema lx = new Lexema();
        lx.setNombreToken(lista2.get(i).getLexema());
        lx.setLexema(lista2.get(i + 1).getLexema());
        if (lista2.get(i + 2).getLexema().equals("=")) {
            lx.setRenglon(1);
        } else {
            lx.setRenglon(0);
        }
        arrLex.add(lx);
    }
}
```

Figura 38 Método optimizaAsignacionesInnecesarias parte 1

En la Figura 39, se vuelve a recorrer la lista en busca de la variable que fue declarada sin valor para eliminar su declaración (tipo de dato, el nombre de variable, el punto y coma). Los datos de su declaración se guardan en variables temporales.

```
//En las variables que se inicializan despues de ser declaradas, poner en su asignacion la declaracion.
//El tipo antes y eliminar su declaracion;
for (int i = 0; i < arrLex.size(); i++) {
    for (int j = 0; j < lista2.size(); j++) {
        if (arrLex.get(i).getLexema().equals(lista2.get(j).getLexema())) {
            if (arrLex.get(i).getRenglon() == 0) {
                Lexema aux = new Lexema();
                aux.setLexema(arrLex.get(i).getLexema());
                aux.setNombreToken(arrLex.get(i).getNombreToken());
                aux.setNumToken(arrLex.get(i).getNumToken());
                aux.setRenglon(1);
                if (lista2.get(j - 1).getLexema().equals("int") || lista2.get(j - 1).getLexema().equals("float")
                    || lista2.get(j - 1).getLexema().equals("boolean") || lista2.get(j - 1).getLexema().equals("String")) {
                    //Eliminar
                    lista2.remove(j - 1); //tipo
                    lista2.remove(j - 1); //lexema
                    lista2.remove(j - 1); //;
                }
            }
        }
    }
}
```

Figura 39 Método optimizaAsignacionesInnecesarias parte 2

En la Figura 40 se recorre la lista en busca de la variable. Si la encuentra acomoda la lista en forma de declaración y termina el ciclo para evitar el reemplazo de las siguientes asignaciones y que ocurra un error semántico, por duplicidad de variables.

```

    for (int i = 0; i < arrLex.size(); i++) {
        for (int j = 0; j < lista2.size(); j++) {
            if (arrLex.get(i).getLexema().equals(lista2.get(j).getLexema())) {
                if (arrLex.get(i).getRenglon() == 0) {
                    Lexema aux = new Lexema();
                    aux.setLexema(arrLex.get(i).getNombreToken());
                    aux.setNombreToken("Tipo de dato");
                    aux.setNumToken(1);
                    lista2.add(j, aux);
                    arrLex.get(i).setRenglon(1);
                    break;
                }
            }
        }
    }
    return lista2;
}

```

Figura 40 Método optimizaAsignacionesInnecesarias parte 3

En la Figura 41, se muestra el método que optimiza los comentarios, este método recibe una lista vacía, después recorre la lista que contiene todos los tokens del código fuente, si encuentra un comentario tanto de línea como de bloque, este no lo toma, lo que no son comentarios los agrega a la nueva lista y la retorna.

```

public ArrayList< analisis.Lexema > optimizaComentarios(ArrayList< analisis.Lexema > lista2) {
    for (int j = 0; j < lexemas.size(); j++) {
        if ((lexemas.get(j).getNumToken() == 59) || (lexemas.get(j).getNumToken() == 60)) {
        } else {
            analisis.Lexema lx = new analisis.Lexema();
            lx.setLexema(lexemas.get(j).getLexema());
            lx.setNombreToken(lexemas.get(j).getNombreToken());
            lx.setNumToken(lexemas.get(j).getNumToken());
            lx.setRenglon(lexemas.get(j).getRenglon());
            lista2.add(lx);
        }
    }
    return lista2;
}

```

Figura 41 Método optimizaCommetarios

En la Figura 42, se muestra el método que imprime la optimización en el compilador, para ello primero se acomoda el código, en forma lineal, es decir, si llega un punto y coma o una llave que abre o cierra, entonces hace un salto de línea, de esta manera no se muestran los tabuladores y los espacios en blanco no son considerados al momento de la impresión.

```
public void imprimeOptimizacion(ArrayList< analisis.Lexema> lista2) {
    String optimizado = "";
    for (int i = 0; i < lista2.size(); i++) {
        optimizado += lista2.get(i).getLexema();
        if (lista2.get(i).getNumToken() == 9 || lista2.get(i).getNumToken() == 10 || lista2.get(i).getNumToken() == 2) {
            optimizado += "\n";
        } else if (lista2.get(i).getNumToken() == 1 || lista2.get(i).getNumToken() == 16
            || lista2.get(i).getNumToken() == 19 || lista2.get(i).getNumToken() == 21
            || lista2.get(i).getNumToken() == 56) {
            optimizado += " ";
        }
    }
    Codigo.optimizado.setText(optimizado);
}
```

Figura 42 *imprimeOptimizacion*

En la Figura 43, se muestra el método que resuelve de manera definitiva las operaciones aritméticas, relaciones y lógicas con el ultimo valor de las variables ocupadas, para ello se revisó la tabla de símbolos. Una vez hecho esto, se remueve cada operando y operador que fue ocupado para su resolución y se sustituyo por el resultado de la operación.

```
private ArrayList<Lexema> optimizaOperaciones(ArrayList<Lexema> lista2) {
    for (int i = 0; i < lista2.size(); i++) {
        if (lista2.get(i).getNumToken() == 53) {
            i++;
            int renglon = 0;
            ArrayList<String> arrInt = new ArrayList<>();
            boolean bandera = true;
            while (bandera) {
                if (lista2.get(i).getNumToken() == 2 || lista2.get(i).getNumToken() == 15) {
                    bandera = false;
                } else {
                    if (lista2.get(i).getNumToken() == 50) {
                        for (int j = 0; j < tablaSimbolos.size(); j++) {
                            if (lista2.get(i).getLexema().equals(tablaSimbolos.get(j).getVariable())) {
                                arrInt.add(tablaSimbolos.get(j).getValor());
                                break;
                            }
                        }
                    } else {
                        arrInt.add(lista2.get(i).getLexema());
                    }
                    renglon = lista2.get(i).getRenglon();
                    lista2.remove(i);
                }
            }
            String valor = Semantico.conversionArrayCola(arrInt); //RETORNA EL VALOR DE LA OPERACION
            // String tipoDatoValor = Semantico.tipoDato(valor); //RETORNA EL TIPO DE DATO DEL VALOR
            Lexema ls = new Lexema();
            ls.setLexema(valor);
            ls.setNombreToken("Operacion");
            ls.setRenglon(renglon);
            ls.setNumToken(200);
            lista2.add(i, ls);
        }
    }
    return lista2;
}
```

Figura 43 Método *optimizaOperaciones()*

En la Figura 44, se muestra el método que optimiza las operaciones, pero sin tomar en cuenta las variables, es decir, si existe una operación: $\text{var} * 9 - 8 + 76 * 2$ entonces solo se realizara: $\text{var} * 153$. Esto es para simplificación de operaciones de mayor costo a menor costo. Entonces se crea un ArrayList para la resolución de operaciones, y otra que almacenara los index del ArrayList lista 2 para saber en dónde se va a ubicar el nuevo valor.

```
public ArrayList<Lexema> optimizarVariablesOp(ArrayList<Lexema> lista2) {
    //Repaso 1: Solo operables
    for (int i = 0; i < lista2.size(); i++) {
        if (lista2.get(i).getNumToken() == 53) {
            int index = i + 1;
            ArrayList<Integer> indexs = new ArrayList<>();
            boolean n = true;
            ArrayList<String> str = new ArrayList<>();
            int k = i + 1;
            while (lista2.get(k).getNumToken() != 2 && n) {
                if (lista2.get(k).getNumToken() == 50) {
                    n = false;
                    break;
                } else {
                    str.add(lista2.get(k).getLexema());
                    indexs.add(k);
                }
                k++;
            }
        }
    }
}
```

Figura 44 Método optimizaVariableOp() parte 1

En la Figura 45, se retorna el resultado en la variable res y su tipo de dato, para sustituir el lexema en la lista 2, también se hace un switch-case para saber qué tipo de dato es, y convertirlo a su número de token, al final se agrega en el índice especificado, y se limpia la lista de índices.

```
if (n == true) {
    String res = Semantico.conversionArrayCola(str);
    String tipo = Semantico.tipoDato(res);
    for (int j = 0; j < indexs.size() - 1; j++) {
        lista2.remove(indexs.get(j));
    }
    Lexema ls = new Lexema();
    ls.setLexema(res);
    switch (tipo) {
        case "INTEGER":
            ls.setNumToken(51);
            break;
        case "FLOAT":
            ls.setNumToken(52);
            break;
        case "BOOLEAN":
            if (res.equals("true")) {
                ls.setNumToken(22);
            } else {
                ls.setNumToken(23);
            }
            break;
        default:
            ls.setNumToken(53);
            break;
    }
    lista2.set(indexs.get(indexs.size() - 1), ls);
    indexs.clear();
}
}
```

Figura 45 Método optimizaVariableOp() parte 2

En la Figura 46, dado a que los procesos anteriores, la expresión puede quedar como: **var + 0 + 9 +10 * 59 + var2**. Primero se revisa si hay un + o un -, después si antes hay un =, si no lo hay entonces pregunta si lo que hay antes se trata de una variable, o una cadena, o un paréntesis que cierra o que abre. Si es así entonces se va a la derecha a comparar si lo que se trata en dos posiciones adelante es un numero o un punto y coma, o una coma. Para eliminar operaciones entre variables de suma o resta.

```
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getLexema().equals("+") || lista2.get(i).getLexema().equals("-")) {
        int index2 = i - 1;
        //Hacia la izquierda
        if (lista2.get(i - 1).getNumToken() == 50) {
        } else {
            if (lista2.get(i - 2).getNumToken() == 53 || lista2.get(i - 2).getNumToken() == 7
                || lista2.get(i - 2).getLexema().equals("+") || lista2.get(i - 2).getLexema().equals("-")) {
                //Hacia la derecha
                if (lista2.get(i + 1).getNumToken() == 50) {
                } else {
                    if (lista2.get(i + 2).getNumToken() == 15 || lista2.get(i + 2).getNumToken() == 8
                        || lista2.get(i + 2).getNumToken() == 2 || lista2.get(i + 2).getLexema().equals("+")
                        || lista2.get(i + 2).getLexema().equals("-")) {
                        //Si
                        int op1 = Integer.parseInt(lista2.get(i - 1).getLexema());
                        int op2 = Integer.parseInt(lista2.get(i + 1).getLexema());
                        int result;
                        if (lista2.get(i).getLexema().equals("+")) {
                            result = op1 + op2;
                        } else {
                            result = op1 - op2;
                        }
                        Lexema ls2 = new Lexema();
                        ls2.setLexema(String.valueOf(result));
                        ls2.setNumToken(51);
                        lista2.set(index2, ls2);
                        lista2.remove(index2 + 1);
                        lista2.remove(index2 + 1);
                        i = 0;
                    } else {
                }
            }
        }
    }
}
```

Figura 46 Método optimizaVariablesOp() parte 3

En la figura 47, al igual que la Figura 19, se trata de un análisis de atrás y hacia adelante para saber si la operación que se va a realizar afecta el sentido de la operación, y así poderla resolver o no. Esta iteración se hace hasta que no quede más que analizar. Este ciclo es para la resolución entre operandos entre variables para la multiplicación y división.

```
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getLexema().equals("*") || lista2.get(i).getLexema().equals("/")) {
        int index2 = i - 1;
        //Hacia la izquierda
        if (lista2.get(i - 1).getNumToken() == 50) {
        } else {
            if (lista2.get(i - 2).getNumToken() == 53 || lista2.get(i - 2).getNumToken() == 7
                || lista2.get(i - 2).getLexema().equals("+") || lista2.get(i - 2).getLexema().equals("-")
                || lista2.get(i - 2).getLexema().equals("/") || lista2.get(i - 2).getLexema().equals("*")) {
                //Hacia la derecha
                if (lista2.get(i + 1).getNumToken() == 50) {
                } else {
                    if (lista2.get(i + 2).getNumToken() == 15 || lista2.get(i + 2).getNumToken() == 8
                        || lista2.get(i + 2).getNumToken() == 2 || lista2.get(i + 2).getLexema().equals("+")
                        || lista2.get(i + 2).getLexema().equals("-") || lista2.get(i + 2).getLexema().equals("/")
                        || lista2.get(i + 2).getLexema().equals("*")) {
                        //Si
                        int op1 = Integer.parseInt(lista2.get(i - 1).getLexema());
                        int op2 = Integer.parseInt(lista2.get(i + 1).getLexema());
                        int result;
                        if (lista2.get(i).getLexema().equals("*")) {
                            result = op1 * op2;
                        } else {
                            result = op1 / op2;
                        }
                        Lexema ls2 = new Lexema();
                        ls2.setLexema(String.valueOf(result));
                        ls2.setNumToken(51);
                        lista2.set(index2, ls2);
                        lista2.remove(index2 + 1);
                        lista2.remove(index2 + 1);
                        i = 0;
                    } else {
                }
            }
        }
    }
}
```

Figura 47 Método optimizaVariablesOp() parte 4

En la Figura 48, se muestra el método que reestructura el código fuente, primero captura el índice que le sigue a la palabra reservada `clase`, después se crean dos listas. Una para las variables `lxV`, y otra para los métodos `lxM`, se itera la `lista2` para encontrar los métodos estos empiezan con la palabra reservada `void` y terminan con una llave que cierra, para controlar las llaves que abren y cierran dentro del método se crea una variable de control que verificara cuantas llaves que abren, y cierran. Cuando ya no encuentra ninguna que se corresponda, entonces termina de agregar los lexemas que contiene el método.

```
public ArrayList<Lexema> optimizarEstructura(ArrayList<Lexema> lista2) {  
    //Variables globales  
    int renglonClase = 0;  
    for (int i = 0; i < lista2.size(); i++) {  
        if (lista2.get(i).getNumToken() == 55) {  
            if (i + 1 < lista2.size()) {  
                if (lista2.get(i + 1).getNumToken() == 9) {  
                    renglonClase = i + 2;  
                }  
            }  
        }  
    }  
    ArrayList<Lexema> lxV = new ArrayList<>();  
    ArrayList<Lexema> lxM = new ArrayList<>();  
    for (int i = 0; i < lista2.size(); i++) {  
        if (lista2.get(i).getNumToken() == 21) {  
            //Metodos  
            int llaveAbre = 0;  
            boolean llaveCierra = false;  
            while (true) {  
                if (lista2.get(i).getNumToken() == 9) {  
                    if (!llaveCierra) {  
                        llaveCierra = true;  
                    }  
                    llaveAbre++;  
                } else if (lista2.get(i).getNumToken() == 10) {  
                    llaveAbre--;  
                }  
                Lexema lx = new Lexema();  
                lx.setLexema(lista2.get(i).getLexema());  
                lx.setNumToken(lista2.get(i).getNumToken());  
                lx.setNombreToken(lista2.get(i).getNombreToken());  
                lxM.add(lx);  
                lista2.remove(i);  
                if (llaveAbre == 0 && llaveCierra) {  
                    break;  
                }  
            }  
        }  
    }  
}
```

Figura 48 Método `optimizaEstructura()` parte 1

En la Figura 49, se itera la lista 2, una vez que se han quitado todos los métodos, en la lista 2 quedarán solo las declaraciones y asignaciones de variables globales por lo que solo se agregan a la lista de variables.

En el penúltimo for, se agregan primero a la lista2 las variables globales, teniendo en cuenta el indicador del índice al cual se van añadiendo. Después en el último for se agregan los métodos.

```
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getNumToken() == 1 || lista2.get(i).getNumToken() == 50) {
        while (true) {
            if (lista2.get(i).getNumToken() == 2) {
                Lexema lx = new Lexema();
                lx.setLexema(lista2.get(i).getLexema());
                lx.setNumToken(lista2.get(i).getNumToken());
                lx.setNombreToken(lista2.get(i).getNombreToken());
                lxV.add(lx);
                lista2.remove(i);
                i--;
                break;
            } else {
                Lexema lx = new Lexema();
                lx.setLexema(lista2.get(i).getLexema());
                lx.setNumToken(lista2.get(i).getNumToken());
                lx.setNombreToken(lista2.get(i).getNombreToken());
                lxV.add(lx);
                lista2.remove(i);
            }
        }
    }
}

for (int i = 0; i < lxV.size(); i++) {
    lista2.add(renglonClase, lxV.get(i));
    renglonClase++;
}

for (int i = 0; i < lxM.size(); i++) {
    lista2.add(renglonClase, lxM.get(i));
    renglonClase++;
}

return lista2;
}
```

Figura 49 Método *optimizaEstructura()* parte 2

En la Figura 50, se muestra el método que verificara si una variable está siendo ocupada por otra variable. Se crea un ArrayList de tipo Variable que guardara, el nombre de la variable, si está declarada, si esta usada, y si esta inicializada.

Esta lista se llenará a cuando en la iteración de la lista2 se encuentre la declaración. Y si esta es inicializada, de una vez será actualizado el registro de la variable.

```
private ArrayList<Lexema> optimizaVariablesSinUso(ArrayList<Lexema> lista2) {
    ArrayList<Variable> arrV = new ArrayList<>();
    for (int i = 0; i < lista2.size(); i++) {
        if (lista2.get(i).getNumToken() == 1) {
            i++;
            if (lista2.get(i).getNumToken() == 50) {
                Variable v = new Variable(lista2.get(i).getLexema(), null, false, false, 0, null);
                i++;
                if (lista2.get(i).getNumToken() == 53) {
                    v.setInicializada(true);
                }
                arrV.add(v);
            }
        }
    }
}
```

Figura 50 Método optimizaVariablesSinUso() parte 1

En la Figura 50, se hace una optimización, esta es la transformación algebraica, y dice que si una variable esta multiplicada o dividida por 1, entonces el resultado será la misma variable, si esta es multiplicada por 0, el resultado será 0, tanto si esto se encuentra a la izquierda de la variable o a su derecha, al igual con la suma, si se encuentra una suma o resta con el número 0, si es así entonces se eliminan dos elementos de la lista2, el operador y el elemento neutro.

```
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getNumToken() == 50) {
        if (lista2.get(i - 1).getLexema().equals("*")) {
            if (lista2.get(i - 2).getLexema().equals("1")) {
                lista2.remove(i - 2);
                lista2.remove(i - 2);
            } else if (lista2.get(i - 2).getLexema().equals("0")) {
                lista2.remove(i - 1);
                lista2.remove(i);
            }
        } else if (lista2.get(i + 1).getLexema().equals("/") || lista2.get(i + 1).getLexema().equals("/")) {
            if (lista2.get(i + 2).getLexema().equals("1")) {
                lista2.remove(i + 1);
                lista2.remove(i + 1);
            } else if (lista2.get(i + 1).getLexema().equals("*") && lista2.get(i + 2).getLexema().equals("0")) {
                lista2.remove(i + 1);
                lista2.remove(i);
            }
        } else if (lista2.get(i - 1).getLexema().equals("+") || lista2.get(i - 1).getLexema().equals("-")) {
            if (lista2.get(i - 2).getLexema().equals("0")) {
                lista2.remove(i - 2);
                lista2.remove(i - 2);
            }
        } else if (lista2.get(i + 1).getLexema().equals("+") || lista2.get(i + 1).getLexema().equals("-")) {
            if (lista2.get(i + 2).getLexema().equals("0")) {
                lista2.remove(i + 1);
                lista2.remove(i + 1);
            }
        }
    }
}
```

Figura 51 Método optimizaVariablesSinUso() parte 2

En la Figura 52, se realiza una optimización, esta se trata de las redundancias que pueden haber por ejemplo: `String var=var;` o `num1=num1;`

Se itera la `lista2`, y busca la redundancia, si la encuentra, condiciona si es una declaración o una asignación. Si es una declaración elimina desde el tipo de dato hasta el punto y coma, si no lo es solo elimina, la variable, el igual, la misma variable y el punto y coma.

En el siguiente `for`, se itera la `lista2`, en busca de que la variable se encuentre después de un `=` y antes de un punto y coma, si es así se le asigna en el `setUnica(true)`.

```
for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getNumToken() == 50 && lista2.get(i + 1).getLexema().equals("=")
        && lista2.get(i + 2).getLexema().equals(lista2.get(i).getLexema())
        && lista2.get(i + 3).getLexema().equals(";")) {
        if (lista2.get(i).getNumToken() == 1) {
            lista2.remove(i - 1);
            lista2.remove(i - 1);
            lista2.remove(i - 1);
            lista2.remove(i - 1);
            lista2.remove(i - 1);
        } else {
            lista2.remove(i);
            lista2.remove(i);
            lista2.remove(i);
            lista2.remove(i);
        }
    }
}

for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getLexema().equals("=")) {
        i++;
        for (int j = 0; j < arrV.size(); j++) {
            if (lista2.get(i - 2).getLexema().equals(arrV.get(j).getVariable())) {
                arrV.get(j).setInicializada(true);
            }
        }
        while (!lista2.get(i).getLexema().equals(";")) {
            if (lista2.get(i).getNumToken() == 50) {
                for (int j = 0; j < arrV.size(); j++) {
                    if (lista2.get(i).getLexema().equals(arrV.get(j).getVariable())) {
                        arrV.get(j).setUnica(true);
                    }
                }
            }
            i++;
        }
    }
}
```

Figura 52 Método `optimizaVariablesSinUso()` parte 3

En la Figura 53, se optimiza las variables que fueron declaradas, pero no tienen en ningún momento del código un valor.

Se elimina la variable en los siguientes casos, si está declarada de manera individual, si esta después del tipo, pero si hay otras variables que también se declaran en la misma línea, si está declarada en medio de otras variables por comas, o si esta al final de las declaraciones. Se itera hasta que no se encuentre ninguna variable sin valor.

```

for (int i = 0; i < lista2.size(); i++) {
    if (lista2.get(i).getNumToken() == 50) {
        for (int j = 0; j < arrV.size(); j++) {
            if (lista2.get(i).getLexema().equals(arrV.get(j).getVariable()) && (arrV.get(j).isInicializada() == false)) {
                if (lista2.get(i - 1).getNumToken() == 1) {
                    if (lista2.get(i + 1).getNumToken() == 2) {
                        //Se elimina el tipo, lexema, punto y coma
                        lista2.remove(i - 1);
                        lista2.remove(i - 1);
                        lista2.remove(i - 1);
                    } else if (lista2.get(i + 1).getNumToken() == 15) { //coma
                        //Se elimina el lexema, coma
                        lista2.remove(i + 1);
                        lista2.remove(i + 1);
                    }
                } else if (lista2.get(i - 1).getNumToken() == 15) {
                    if (lista2.get(i + 1).getNumToken() == 2) {
                        //Se elimina la coma , lexema
                        lista2.remove(i - 1);
                        lista2.remove(i - 1);
                    } else if (lista2.get(i + 1).getNumToken() == 15) { //coma
                        //Se elimina el coma, lexema
                        lista2.remove(i - 1);
                        lista2.remove(i - 1);
                    }
                }
                i = 0;
            } else {
            }
        }
    }
}

```

Figura 53 Método *optimizaVariablesSinUso()* parte 4

En la Figura 54, se eliminan las variables que, si fueron inicializadas o fueron asignadas, pero nunca fueron usadas por otras variables. Se elimina la variable en cada momento en que fue asignada. Se realiza el while hasta que no encuentre más rastro de la variable y termina.

```
//Variables que tienen valor pero nunca son usadas
for (int i = 0; i < arrV.size(); i++) {
    boolean valor = false;
    for (int j = 0; j < lista2.size(); j++) {
        if (lista2.get(j).getNumToken() == 50) {
            if (arrV.get(i).getVariable().equals(lista2.get(j).getLexema())) {
                if (lista2.get(j + 1).getLexema().equals("=")) {
                    j++;
                    while (!lista2.get(j).getLexema().equals(";")) {
                        if (lista2.get(j).getNumToken() == 50) {
                            valor = true;
                            break;
                        } else {
                            j++;
                        }
                    }
                }
            }
        }
    }
}
}
```

Figura 54 Método optimizaVariablesSinUso() parte 5

En la Figura 55, se elimina la variable que no fue usada, en cada asignación, y desde la declaración, en cada uno de los casos en los que se puede declarar la variable, después de tipo, en medio de otras variables o al final.

int a=0; int a, b; int b, a,c; int b, c, a;

```
if (valor == false && arrV.get(i).isUnica() == false) {
    for (int j = 0; j < lista2.size(); j++) {
        if (lista2.get(j).getNumToken() == 50) {
            if (arrV.get(i).getVariable().equals(lista2.get(j).getLexema())) {
                if (lista2.get(j - 1).getNumToken() == 1) {
                    if (lista2.get(j + 1).getNumToken() == 2) {
                        //Se elimina el tipo, lexema, punto y coma
                        lista2.remove(j - 1);
                        lista2.remove(j - 1);
                        lista2.remove(j - 1);
                    } else if (lista2.get(j + 1).getNumToken() == 15) { //coma
                        //Se elimina el lexema, coma
                        lista2.remove(j + 1);
                        lista2.remove(j + 1);
                    } else if (lista2.get(j + 1).getLexema().equals("=")) {
                        //eliminar
                        j++;
                        while (true) {
                            if (lista2.get(j).getLexema().equals(";") || lista2.get(j).getLexema().equals(",")) {
                                break;
                            } else {
                                lista2.remove(j);
                            }
                        }
                    }
                    //Eliminar
                }
            }
        }
    }
}
```

Figura 55 Método optimizaVariablesSinUso() parte 6

Capítulo VII. Instalación

Para instalar el compilador, seleccionamos el icono como se muestra en la Figura 58, damos clic derecho, y seleccionamos **ejecutar como administrador**.



Figura 58 Ejecutar como administrador

Enseguida nos aparecerá una ventana, en la que se escogerá el idioma de instalación, como se aprecia en la Figura 59, en este caso **Español**, le damos **aceptar**.

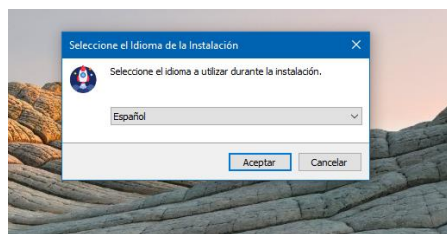


Figura 59 Seleccionar idioma de instalación

A continuación, se deberá seleccionar la carpeta en donde se instalarán los archivos del programa. En la Figura 60, se muestra como predeterminada la ruta **Program Files(x86)**, damos en **Siguiente**.

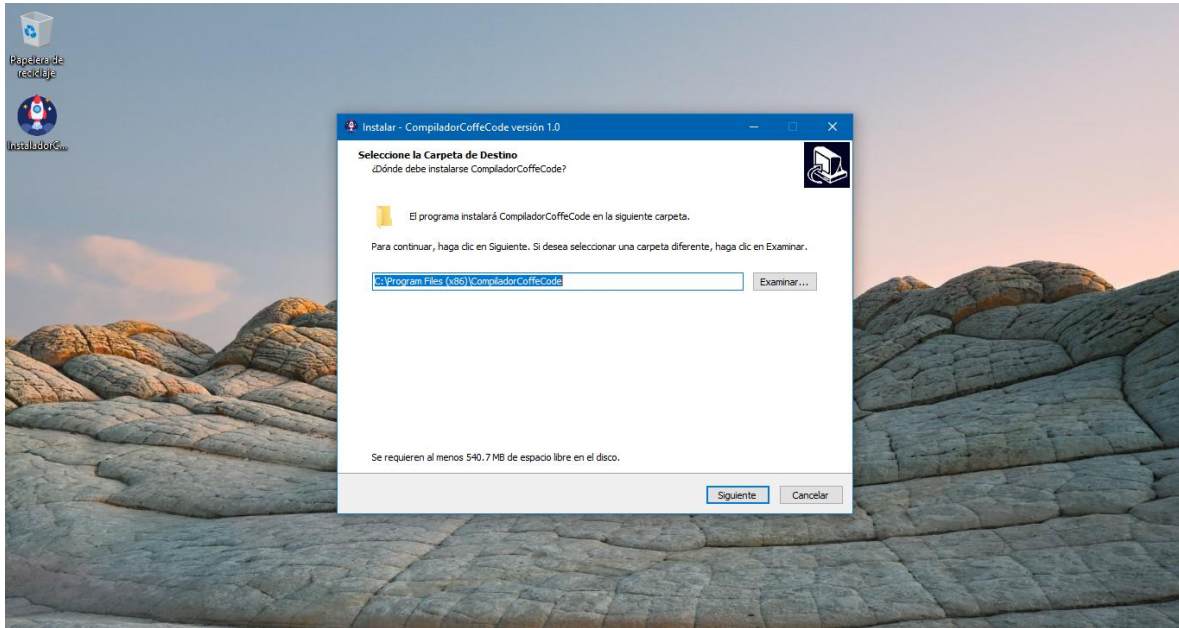


Figura 60 Seleccionar directorio destino de instalación

Enseguida se mostrarán las tareas adicionales, en este caso si deseamos Crear un icono en el escritorio. Le seleccionaremos la casilla y le damos en **Siguiente**. (Figura 61)

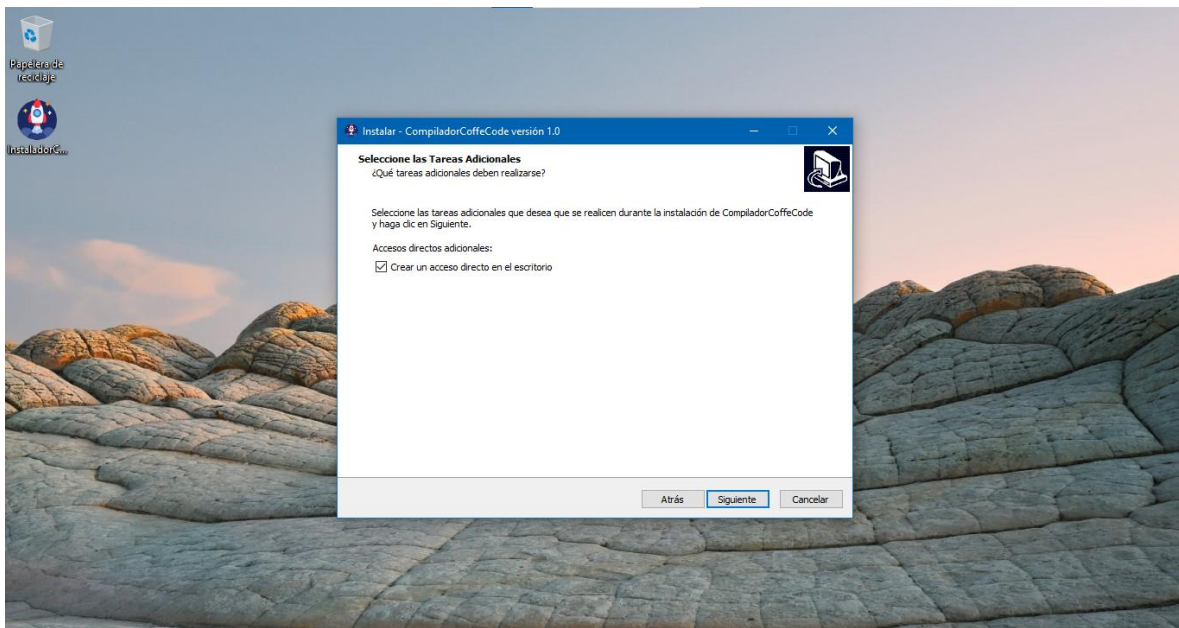


Figura 61 Crear un icono en el escritorio

Por último, se mostrará un resumen de lo que se configuró (Figura 62), y presionamos en Instalar, para vaciar las carpetas y archivos necesarios para el programa.

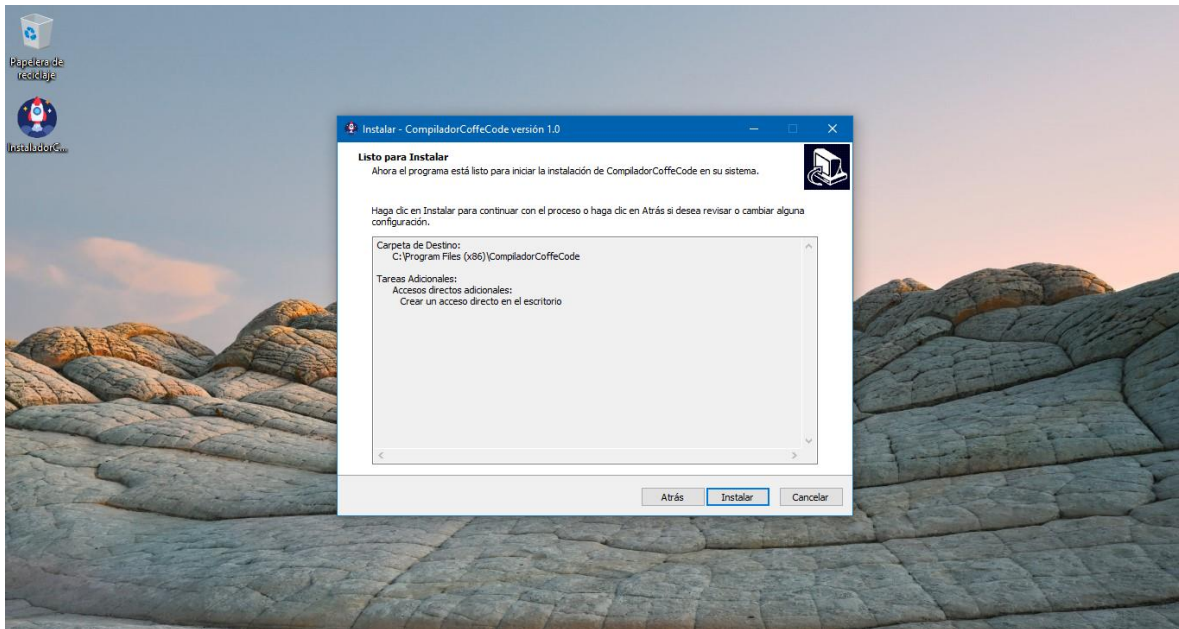


Figura 62 Resumen de configuración

En la Figura 63, se muestra un estado del vaciado de archivos, esperamos a que termine.

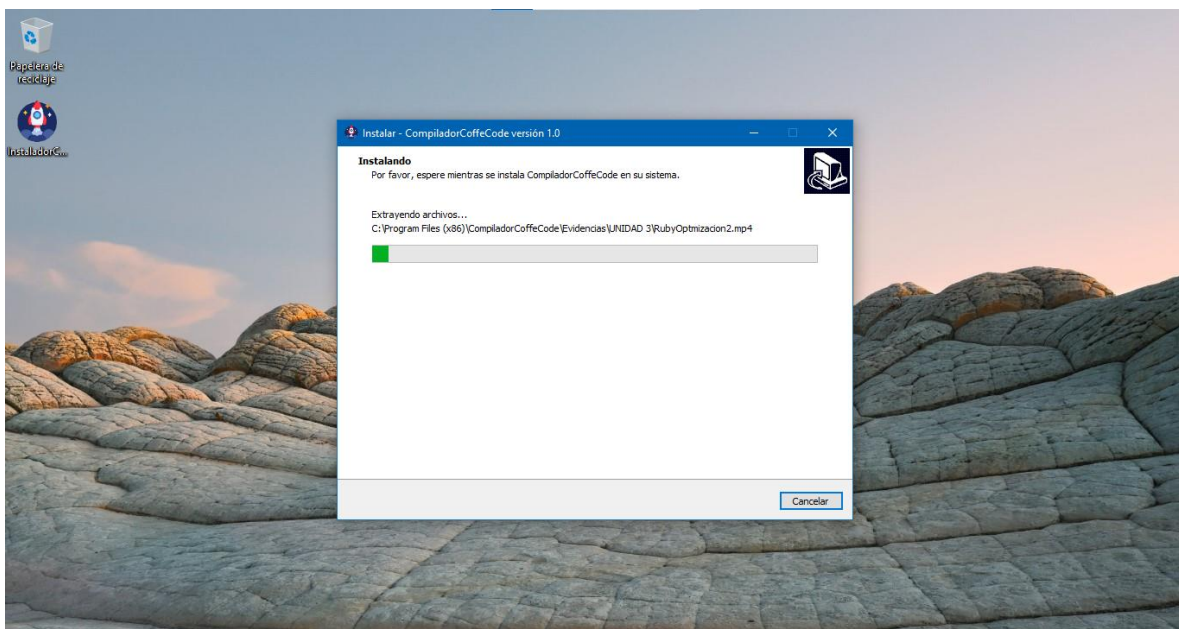


Figura 63 Vaciado de archivos

Finalmente, nos aparece una ventana de que la instalación se completó (Figura 64), opcionalmente se desmarcar la casilla por si no se quiere ejecutar ahora, y le damos en **Finalizar**.

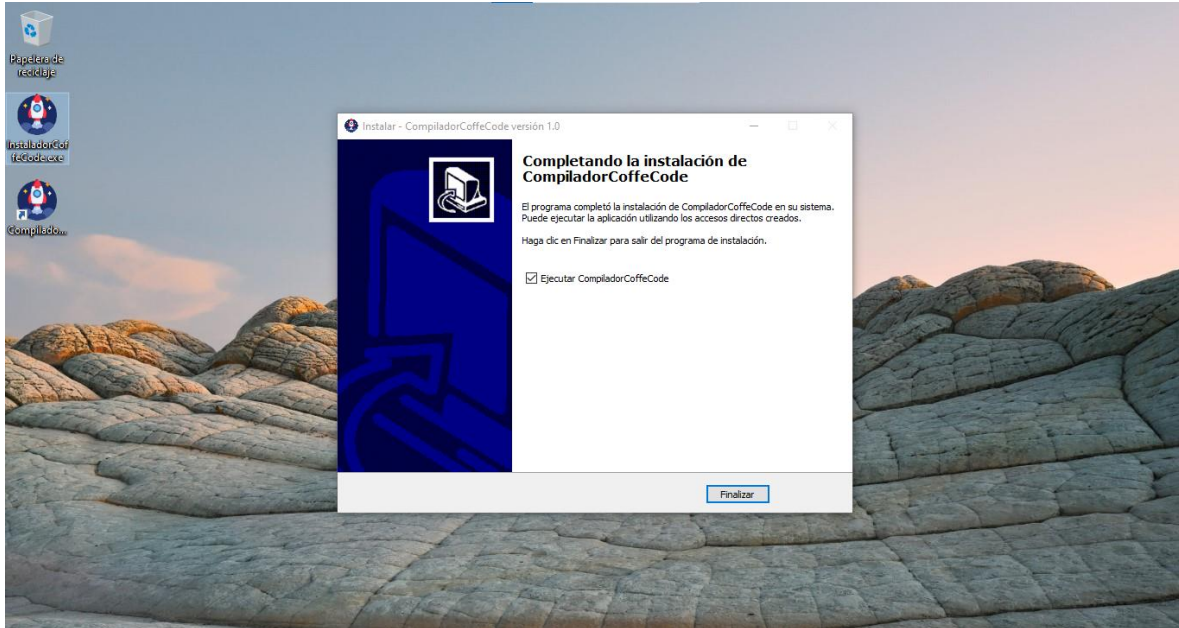


Figura 64 Fin de instalación

En la Figura 65, se muestra corriendo el programa.

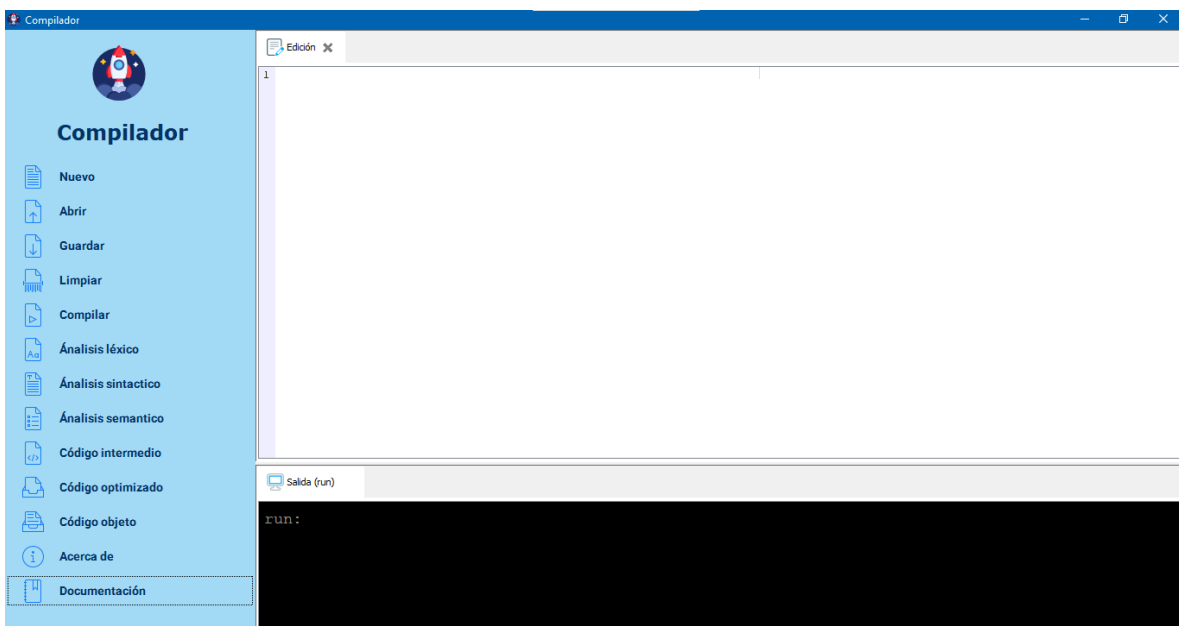


Figura 65 Ejecución del programa

En la Figura 66, se muestra que ya se ha agregado al panel de aplicaciones el programa.

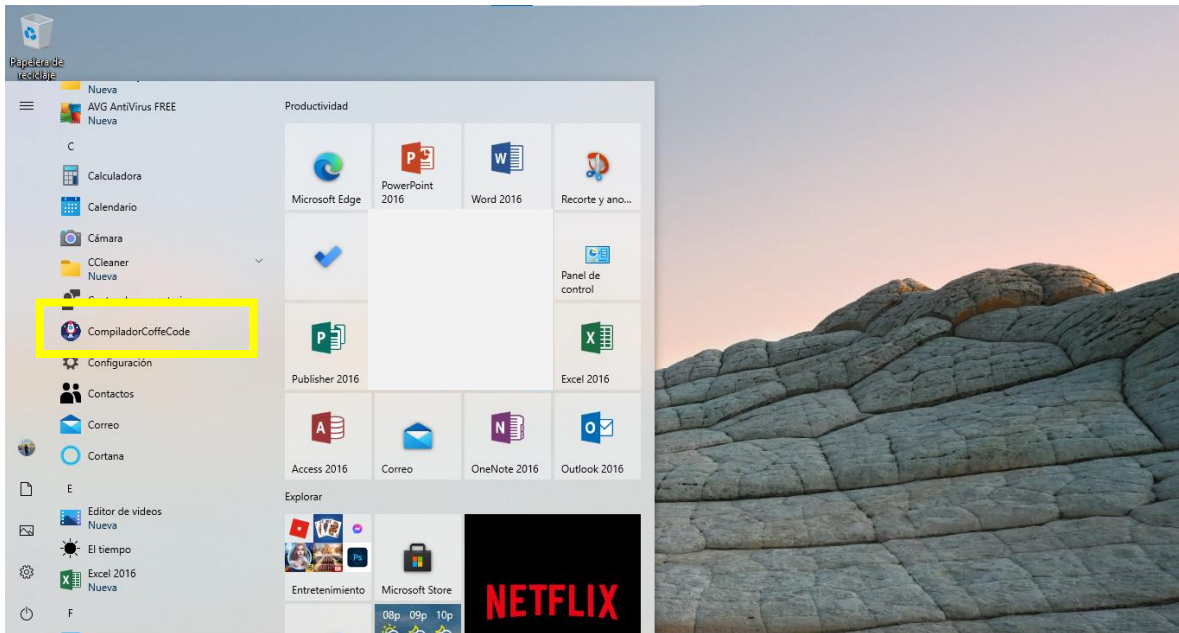


Figura 66 Programa en el panel de aplicaciones

En la Figura 67, se muestran las carpetas y archivos, en dicha ruta.

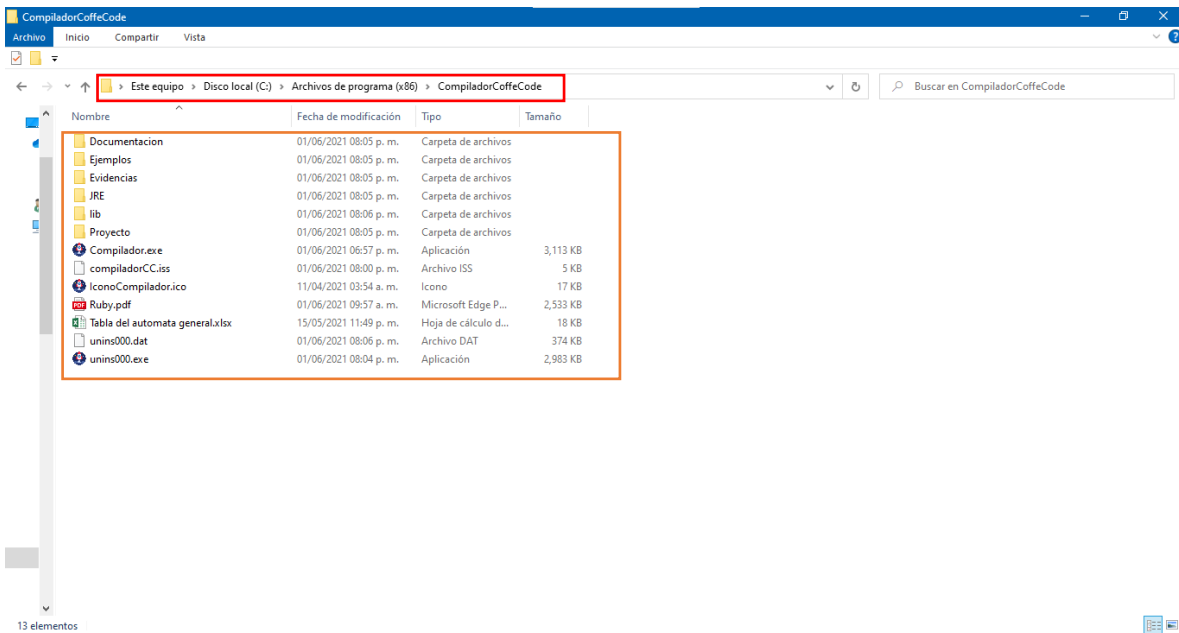


Figura 67 Ruta de archivos

En la Figura 68, se muestra que no existe Java, por lo que se asegura que no es necesario tener el JDK para que el programa pueda ejecutarse correctamente. En la computadora.

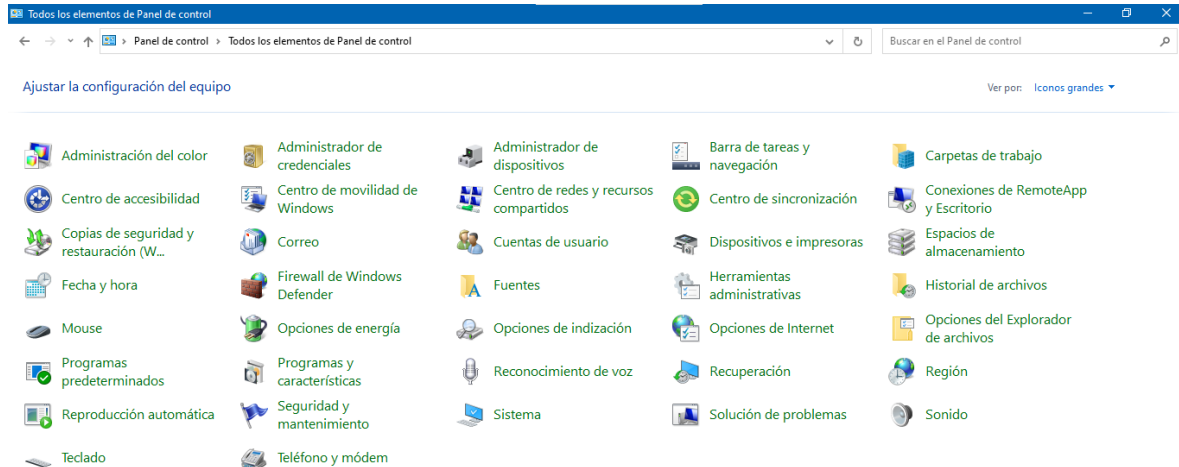


Figura 68 Panel de categorías, sin JAVA

En la Figura 69, se muestra entre el listado de programas el programa recientemente agregado.

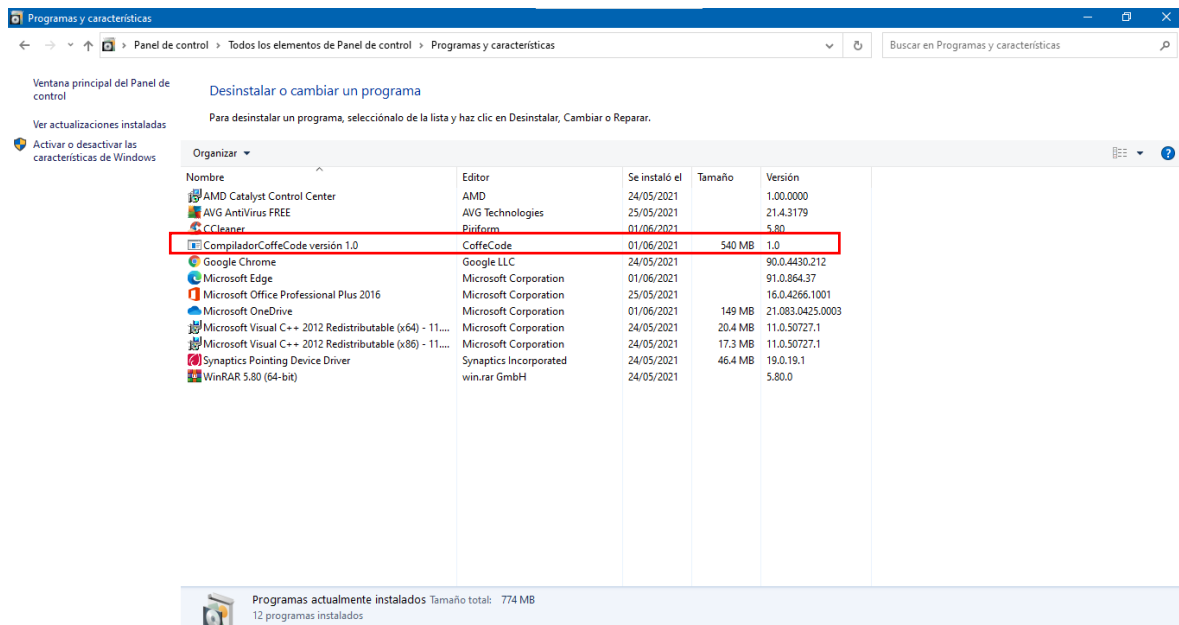


Figura 69 Programa dentro del sistema

Capítulo VIII. Manual de usuario

En la Figura 70 se muestra el entorno de edición del compilador. El cual está compuesto por 3 paneles principales:

- El panel de navegación (Enmarcado con azul)
- El panel de edición (Enmarcado con verde)
- El panel de salida (Enmarcado con amarillo)

Dentro del panel de navegación, se encuentran los siguientes botones:

- Nuevo
- Abrir
- Guardar
- Limpiar
- Compilar
- Análisis léxico
- Análisis sintáctico
- Análisis semántico
- Código intermedio
- Código optimizado

- Código Objeto
- Acerca de
- Documentación

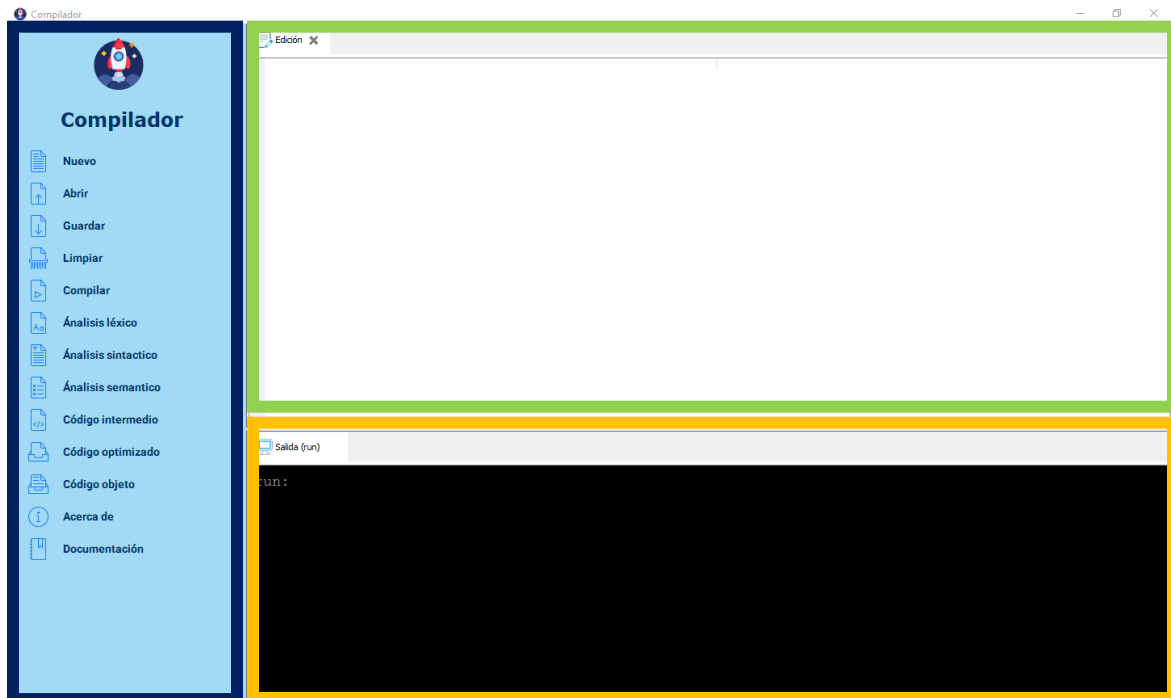


Figura 70 Entorno de edición

Nuevo

Al presionar sobre este botón, se le abra una ventana como se muestra en la Figura 71, esta ventana de navegación le pedirá que ingrese el nombre del nuevo archivo, y también se le pide escoger una carpeta en donde se guardará dicho archivo.

Una vez hecho este paso, usted ya podrá escribir sobre el documento dentro del panel de edición. Esta acción puede realizarse cuantas veces le sea necesario.

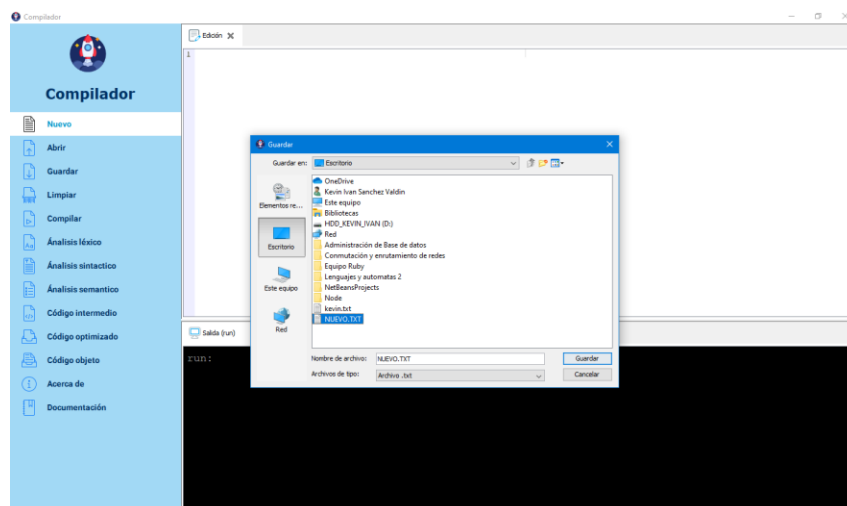


Figura 71 Acción Nuevo

Abrir

Al presionar sobre este botón, se le abra una ventana como se muestra en la Figura 72, en ella, usted navegara por los directorios de su computadora, hasta hallar el archivo que quiera abrir. Al presionar Abrir dentro de la ventana, se cargara el contenido del archivo sobre el panel de edición. Ahora podrá abrir archivos y editarlos en cada uno de ellos.

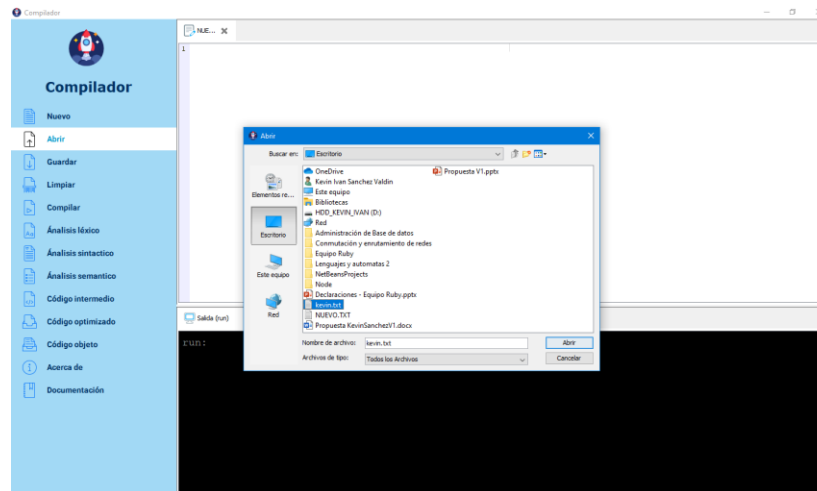


Figura 72 Acción Abrir

Guardar

Al presionar sobre este botón, el archivo del cual se esta seleccionando, se guardará. En este punto, se recomienda presionar este botón si quiere que los cambios hechos en el documento se guarden antes de cerrarlo.

Limpiar

Al presionar sobre este botón, se borrará completamente el contenido del archivo del que se esta trabajando, además, se limpiaran los analisis, tanto lexico, sintactico, semantico, intermedio, optimización y la tabla de simbolos.

Compilar

Al presionar sobre este botón, el proceso de compilación sobre el archivo seleccionado comenzará. Primero se realiza el analisis lexico, seguido del analisis sintactico, semantico, generación del codigo intermedio, generación del codigo optimizado. Al terminar el proceso, se mostrará una ventana que contiene en diferentes paneles los analisis, como se muestran en la Figura 73,



Figura 73 Navegación de los resultados de compilación

En la Figura 73, se muestran por separado las primeras 4 fases de compilación, con su respectivo título: Análisis léxico, análisis sintáctico, análisis semántico. Código intermedio.

Por defecto se muestra el panel del Análisis, sin embargo usted puede consultar las siguientes partes de manera dependiente. En el recuadro rojo, se muestran una navegación entre paneles: Análisis (Opción por defecto), Tabla de símbolos, Código Optimizado.

En la Figura 74, se muestra la Tabla de símbolos, una vez que se presiona en su botón correspondiente.



Figura 74 Panel de la Tabla de símbolos

En la Figura 75, se muestra la generación del Código optimizado, este panel cuenta, con los siguientes apartados, del lado izquierda se encuentra el Código Fuente, y de lado derecho el código optimizado.

- Códigos (Recuadros en color morado)
- Resultados de la optimización (Recuadros en color orquídea)

Arriba del contenedor del código optimización aparece un botón **Reemplazar**, que es el que al presionar pasa el contenido del código optimizado al archivo del que se realizó la optimización.

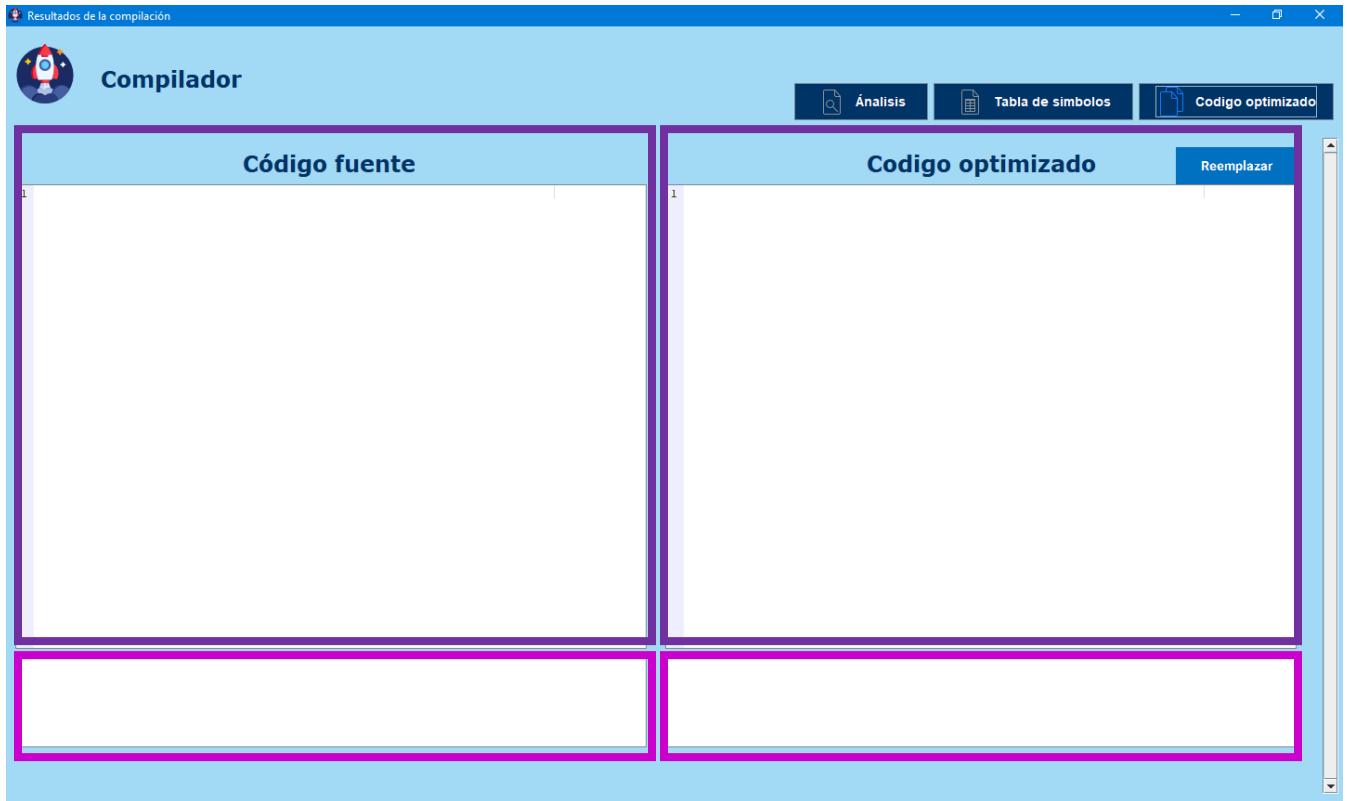


Figura 75 Panel de la generación del código optimizado

Analisis lexico

Al presionar sobre este botón, se realizará el proceso del analisis léxico, del archivo seleccionado. Al terminar, se abra la ventana de resultados, en el contenedor correspondiente como se muestra en la Figura 76.



Figura 76 Contenedor del Análisis léxico

Análisis sintactico

Al presionar sobre este botón, se realizará el proceso del analisis sintactico, del archivo seleccionado. Al terminar, se abrira la ventana de resultados, con el analisis en su correspondiente contenedor, como se muestra en la Figura 77.



Figura 77 Contenedor del Análisis sintáctico

Analisis semantico

Al presionar sobre este botón, se realizará el proceso de analisis semantico correspondientes al codigo fuente seleccionado. Al terminar, se abrira la ventana de resultados, con el analisis en su correspondiente contenedor, como se muestra en la Figura 78.

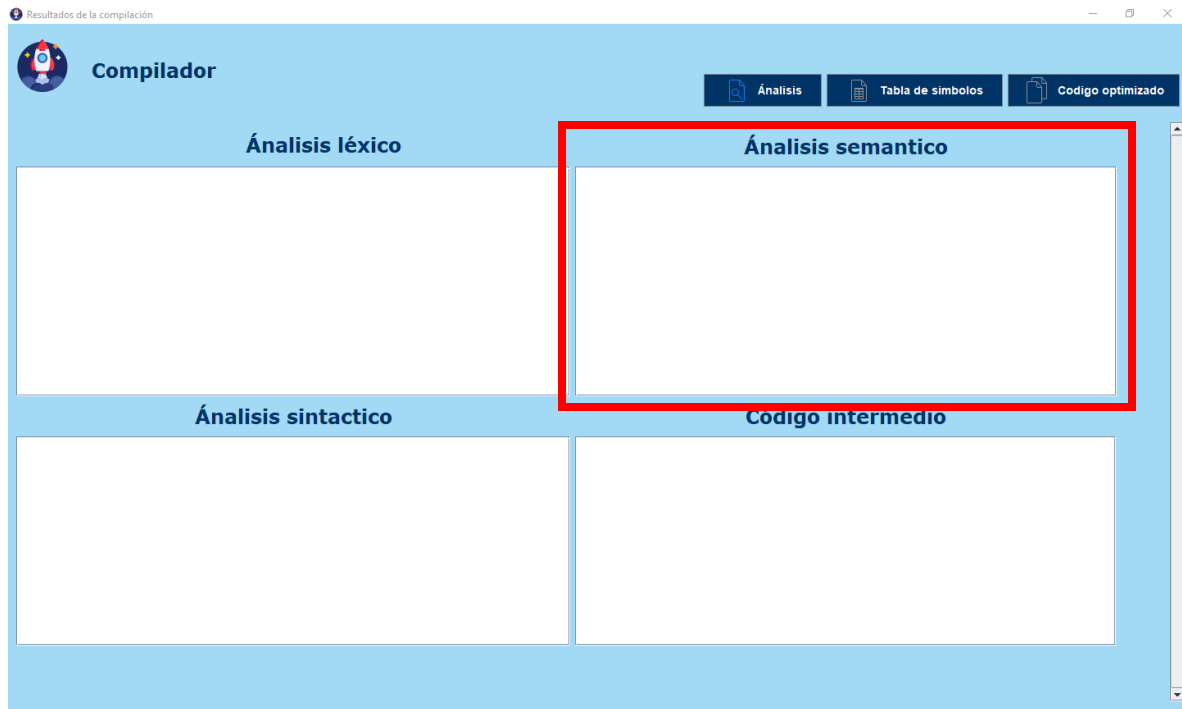


Figura 78 Contenedor del Análisis Semantico

Codigo intermedio

Al presionar sobre este botón, se realizará el proceso la generacion de los cuadрупlos correspondientes a cada operación del archivo fuente, seleccionado. Al terminar, se abrira la ventana de resultados, con el analisis en su correspondiente contenedor, como se muestra en la Figura 79.



Figura 79 Contenedor del Código Intermedio

Codigo optimizado

Al presionar el botón, se comenzará con la generación del código optimizado, este se puede consultar a través del botón Código Optimizado en la Ventana de resultados.

Acerca de

Al presionar sobre este botón, emergerá una ventana que muestra los datos del proyecto. Ver Figura 80.

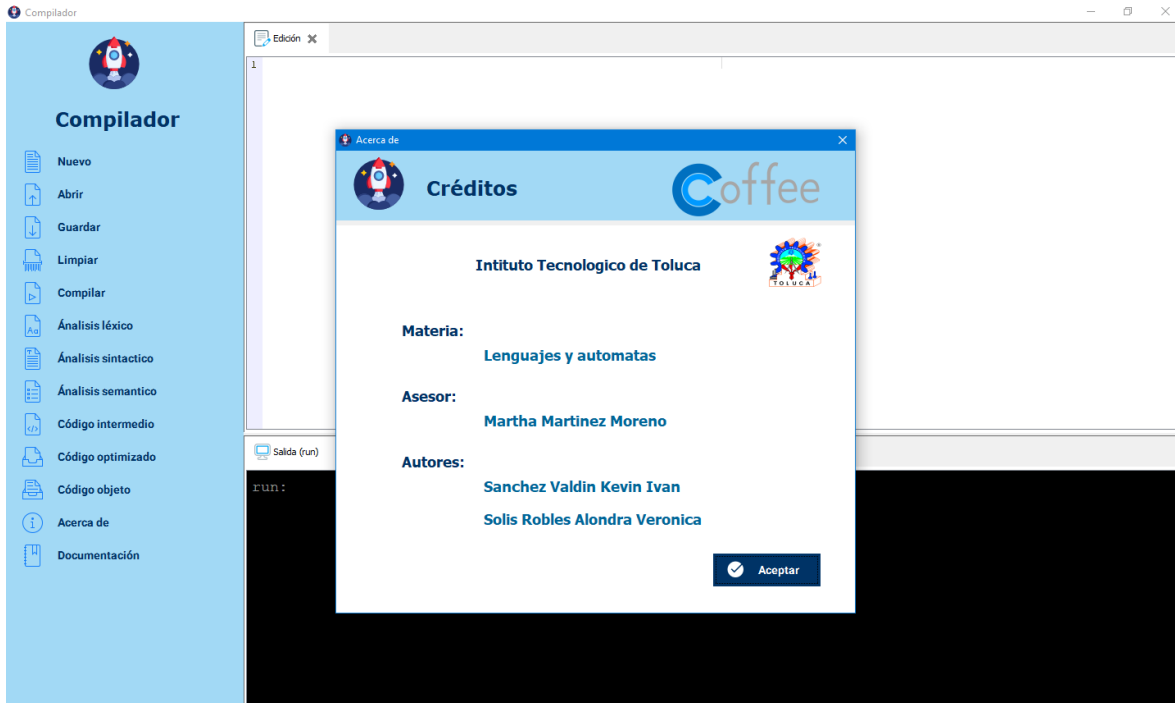


Figura 80 Ventana de Créditos

Documentación

Al presionar sobre este botón, emergerá una ventana que mostrará el documento en el que se encuentra el desarrollo del lenguaje Coffe-Code así como la implementación en el compilador. (Ver Figura 81)

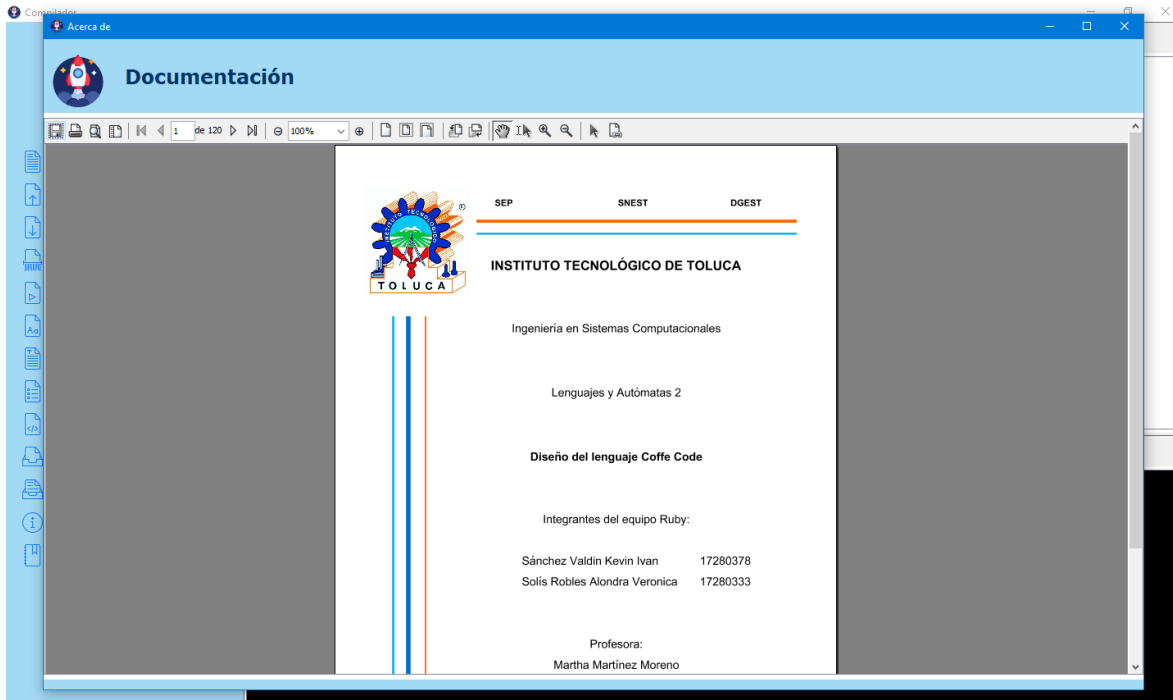


Figura 81 Ventana de la Documentación

Sobre los códigos ejemplo

Los códigos ejemplos se pueden encontrar, apartir de la carpeta en donde se instalo el compilador.

\$CapetaInstalacion > Ejemplos

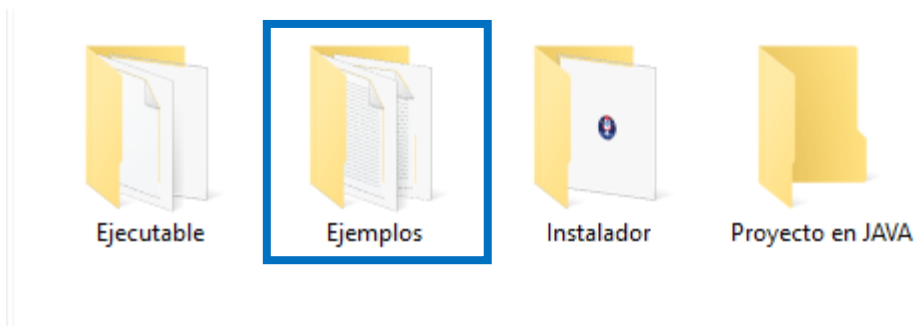


Figura 82 Carpeta de códigos ejemplos

Anexos

Código ejemplo (Análisis léxico)

```
// ***** CODIGO FUENTE *****
// Inicio de la clase
class #ejemplo{
    // Declaracion Y asignacion de variables de variables
    int a = 9+4+10-6-3-30+65;
    int b = 1+2-(3*2)-3+(6+5*5)+4;
    int c = -20 +2;
    int x = 400*20;
    int y = 234+234 + (c*2 /10);
    int af = 20/2-2;
    // Operaciones con variables y numeros
    c = b * 2 + c;
    c = a - b + 3;
    y = y - 400 + 100 * 2;
    // MÃs declaraciones y asignaciones con operaciones
    int var123=10+32+(1+9);
    int var124=(10+(32+1)+7)*6;
    int var125=20+(10*3)+54;
    int var126=30+43*(34-32)-32+43;
    int var127=((40+23)-32)*31-3)-87;
    int var128=((40+23)-32)*31)-3-87+((((((54)+5)-65)*64)-54)+6)+2)+4;
    int var129=(2-(((3*4)+4)-2)+4)-3;
    int var4 = 0*9;
    int a12=9*32-32+3+32*32;
    int a23=9*5-32/3+42-(12*4)+5-32+54;
    int a34=93+243+42*993-43-4+32+32-12+3;
    int b15=a12+a23*var4;
    int b26=a12-43+42+2+42-43;
    int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
    int b48=102*32-31-32/32-21+32;
    int b814= a23*3-32+31+31-42+413-43*43;
    int c98=98*90*190*1902*9;
    int c102=9912*324-332;
    int d5= b15-b26+59;
    //Fin de las de las declaraciones

    // Inicio del Metodo 1
    void _operaciones (){
        // Codigo del metodo 1
        int xxxx = 100*2+4;
        c=1+2-3*2-3;
        while (c > 200) {
            k = x +2;
            if (k > x){
                k = 190;
            } else {
                k = k + 2 * x;
            }
        }
    }
}
```

```
        c = c - 1;
    }
d5 = c*c;
c102 = xxxx - c -d5;

// Comentario ej
y = 234+234;
af = 20/2-2;
x = af+67;
int d = (132+23*(34/3)+234);
int s = d/2;
int ent3 = 10;
int ent7 = 12+14*2;
// Operaciones con variables
x = a +b+c;
d5 = xxxx+35+234*456-345;
a = 35+234*456-345;
int j= 21*3+14-20;
// Operaciones con números
int fg = 20*35+234*456-345;
int u = a+20*35+234*456-345;
a = b *2 *c;
a = 436+235+23+20*35+234*456-345;
b = 400+234;
int k = 234;
b = a*k;
a = a*k+2000;
k = 234+a*c;
// Asignaciones de un solo numero
b = -1;
xxxx = 24;
k = 2000;
// Variables booleanas
boolean bandera = false;
boolean ban = false;
boolean ban1 = false;
boolean ban2 = false;
// Condición if-else
if(a < c){
    x = a+ b;
    a = 35+234*456-345;
} else {
    x = b - a * 100;
    b = a*k;
    a = a*k+2000;
}
x = 180;
// Ciclo While
while (x > 200) {
    k = x +2;
    if (k > x){
        k = 190;
    } else {
        k = k + 2 * x;
    }
}
```

```

        x = x - 1;
    }
    // Condición if
    if (a < b && c < b){
        int aaa=9*5-32/3+42-(12*4)+5-32+54;
        int sss=93+243+42*993-43-4+32+32-12+3;
        int ccc=a12+a23*var4;
        int vvvv=a12-43+42+2+42-43;
        int ttt=100 +31-32+32+32-31+53+432+321+984+90+443;
        int jjj=102*32-31-32/32-21+32;
        int hhh= a23*3-32+31+31-42+413-43*43;
        int uuu=98*90*190*1902*9;
        aaa = 324+23+235;
        uuu = aaa * 2;
        jjj = aaa * uuu;
        jjj = jjj * 2;
        a = 324+23+235;
        b = aaa * 2;
        c = aaa * uuu;
        k = jjj * 2;
    }
}
// Fin del metodo 1
}
// Fin de la clase
// ***** FIN DEL CODIGO FUENTE *****

```

Código ejemplo (Análisis sintáctico)

```

// ***** CODIGO FUENTE *****
// Inicio de la clase
class #ejemplo{
    // Declaracion Y asignacion de variables de variables
    int a = 9+4+10-6-3-30+65;
    int b = 1+2-(3*2)-3+(6+5*5)+4;
    int c = -20 +2;
    int x = 400*20;
    int y = 234+234 + (c*2 /10);
    int af = 20/2-2;
    // Operaciones con variables y numeros
    c = b * 2 + c;
    c = a - b + 3;
    y = y - 400 + 100 * 2;
    // MÃs declaraciones y asignaciones con operaciones
    int var123=10+32+(1+9);
    int var124=(10+(32+1)+7)*6;
    int var125=20+(10*3)+54;
    int var126=30+43*(34-32)-32+43;
    int var127=((40+23)-32)*31-3-87;
    int var128=((40+23)-32)*31-3-87+((((54)+5)-65)*64)-54+6+2+4;
    int var129=(2-((3*4)+4)-2)+4-3;
    int var4 = 0*9;
    int a12=9*32-32+3+32*32;
}

```



```
int a23=9*5-32/3+42-(12*4)+5-32+54;
int a34=93+243+42*993-43-4+32+32-12+3;
int b15=a12+a23*var4;
int b26=a12-43+42+2+42-43;
int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
int b48=102*32-31-32/32-21+32;
int b814= a23*3-32+31+31-42+413-43*43;
int c98=98*90*190*1902*9;
int c102=9912*324-332;
int d5= b15-b26+59;
//Fin de las de las declaraciones

// Inicio del Metodo 1
void _operaciones (){
    //Codigo del metodo 1
    int xxxx = 100*2+4;
    c=1+2-3*2-3;
    while (c > 200) {
        k = x +2;
        if (k > x){
            k = 190;
        } else {
            k = k + 2 * x;
        }
        c = c - 1;
    }
    d5 = c*c;
    c102 = xxxx - c -d5;

    // Comentario ej
    y = 234+234;
    af = 20/2-2;
    x = af+67;
    int d = (132+23*(34/3)+234);
    int s = d/2;
    int ent3 = 10;
    int ent7 = 12+14*2;
    // Operaciones con variables
    x = a +b+c;
    d5 = xxxx+35+234*456-345;
    a = 35+234*456-345;
    int j= 21*3+14-20;
    // Operaciones con números
    int fg = 20*35+234*456-345;
    int u = a+20*35+234*456-345;
    a = b *2 *c;
    a = 436+235+23+20*35+234*456-345;
    b = 400+234;
    int k = 234;
    b = a*k;
    a = a*k+2000;
    k = 234+a*c;
    // Asignaciones de un solo numero
    b = -1;
    xxxx = 24;
```

```
k = 2000;
// Variables booleanas
boolean bandera = false;
boolean ban = false;
boolean ban1 = false;
boolean ban2 = false;
// Condición if-else
if(a < c){
    x = a+ b;
    a = 35+234*456-345;
} else {
    x = b - a * 100;
    b = a*k;
    a = a*k+2000;
}
x = 180;
// Ciclo While
while (x > 200) {
    k = x +2;
    if (k > x){
        k = 190;
    } else {
        k = k + 2 * x;
    }
    x = x - 1;
}
// Condición if
if (a < b && c < b){
    int aaa=9*5-32/3+42-(12*4)+5-32+54;
    int sss=93+243+42*993-43-4+32+32-12+3;
    int ccc=a12+a23*var4;
    int vvvv=a12-43+42+2+42-43;
    int ttt=100 +31-32+32+32-31+53+432+321+984+90+443;
    int jjj=102*32-31-32/32-21+32;
    int hhh= a23*3-32+31+31-42+413-43*43;
    int uuu=98*90*190*1902*9;
    aaa = 324+23+235;
    uuu = aaa * 2;
    jjj = aaa * uuu;
    jjj = jjj * 2;
    a = 324+23+235;
    b = aaa * 2;
    c = aaa * uuu;
    k = jjj * 2;
}

}
// Fin del metodo 1
}
// Fin de la clase
// ***** FIN DEL CODIGO FUENTE *****
```

Código ejemplo (Análisis semántico)

```
class #ejemplo{

    int a,b,c,d,e;
    String saludo = "Hola mundo";
    boolean bol = true;
    float f = 23+(2*3)/3;
    a = 200;
    bol = false;

    void _caracterNoEsperado(){
        int uno = ;
        int dos = , asf;
        int tres = +;
        int cuatro = &&;
        int cinco = ||;
        int cinco1 = 3+25+||;
        int seis = *;
        int seis1 = 234*4-
        int siete = -;
        int ocho = %;
        int r = (32+43+24)+;
    }

    void _asignacionesNoDeclaradas(){
        nvar2 = 32;
        nvar3 = 32;
        nvar2 = 9+2;
        nvar4 = 43;
        nvar4 = 443 +32;
        nvar4 = "Hola";
        nvar3 = "312432";
    }

    void _yaDefinida(){
        int x =10;
        int x =234;
        String x = "sdf";
        boolean x = false;
        int y = x + 2;
        int y = (32+43+24);
        int y = f*2;
    }

    void _declaracionesMisma(){
        float var = 21;
        float var = 3.2;
        float var = "Hola";
        float var = true;
        float var = false;
        float var = "2.2" + "3.2";
        float var = (32) > 32;
        float var = 212 * 32;
        float var = "Hola " * 32;
        float var = (32.3 + 3.2)* 32+32 +(32-32);
    }
}
```

```
void _asignacionesFloat(){
    float flotante = 21;
    flotante = 3.2;
    flotante = "Hola";
    flotante = true;
    flotante = false;
    flotante = 212 * 32;
    flotante = 4* 32 + (12/3);
    flotante = 12.2+14.6*2;
}

void _asignacionesString(){
    String var9;
    var9 = 342;
    var9 = "Hola mundo";
    var9 = true;
    var9 = 9;
    var9 = 9.0;
    var9 = false;
    var9 = 12.2+14.6*2;
}

void _asignacionesBooleanas(){
    boolean centinela = true;
    centinela = "Hola";
    centinela = 30.2 + 456 + 12.3;
    centinela = 234+234-342;
    centinela = true;
    centinela = 9.9 * 42 > 3;
    centinela = 23 < 2 && 48 < 46;
    centinela = true;
    centinela = false;
}

void _asignacionesInt(){
    int y = 234+234;
    int af = 20/2-2;
    int x = af+67;
    int d = (132+23*(34/3)+234);
    int s = d/2;
    int ent1 = "cadena 1";
    int ent2 = false;
    int ent3 = 10.6;
    int ent4 = 234 > 23;
    int ent5 = 23 < 2 && 48 < 46;
    int ent6 = true && false;
    int ent7 = 12.2+14.6*2;
}
}
```

Código ejemplo (Generación del código intermedio)

```
// ***** CODIGO FUENTE *****
```

```
// Inicio de la clase
class #ejemplo{
  // Declaracion Y asignacion de variables de variables
  int a = 9+4+10-6-3-30+65;
  int b = 1+2-(3*2)-3+(6+5*5)+4;
  int c = -20 +2;
  int x = 400*20;
  int y = 234+234 + (c*2 /10);
  int af = 20/2-2;
  // Operaciones con variables y numeros
  c = b * 2 + c;
  c = a - b + 3;
  y = y - 400 + 100 * 2;
  // MÃs declaraciones y asignaciones con operaciones
  int var123=10+32+(1+9);
  int var124=(10+(32+1)+7)*6;
  int var125=20+(10*3)+54;
  int var126=30+43*(34-32)-32+43;
  int var127=((40+23)-32)*31-3)-87;
  int var128=((40+23)-32)*31)-3-87+((((((54)+5)-65)*64)-54)+6)+2)+4;
  int var129=(2-((3*4)+4)-2)+4)-3;
  int var4 = 0*9;
  int a12=9*32-32+3+32*32;
  int a23=9*5-32/3+42-(12*4)+5-32+54;
  int a34=93+243+42*993-43-4+32+32-12+3;
  int b15=a12+a23*var4;
  int b26=a12-43+42+2+42-43;
  int b37=100 +31-32+32+32-31+53+432+321+984+90+443;
  int b48=102*32-31-32/32-21+32;
  int b814= a23*3-32+31+31-42+413-43*43;
  int c98=98*90*190*1902*9;
  int c102=9912*324-332;
  int d5= b15-b26+59;
  //Fin de las de las declaraciones

  // Inicio del Metodo 1
  void _operaciones (){
    // Codigo del metodo 1
    int xxxx = 100*2+4;
    c=1+2-3*2-3;
    while (c > 200) {
      k = x +2;
      if (k > x){
        k = 190;
      } else {
        k = k + 2 * x;
      }
      c = c - 1;
    }
    d5 = c*c;
    c102 = xxxx - c -d5;

    // Comentario ej
    y = 234+234;
    af = 20/2-2;
  }
}
```

```
x = af+67;
int d = (132+23*(34/3)+234);
int s = d/2;
int ent3 = 10;
int ent7 = 12+14*2;
// Operaciones con variables
x = a +b+c;
d5 = xxxx+35+234*456-345;
a = 35+234*456-345;
int j= 21*3+14-20;
// Operaciones con números
int fg = 20*35+234*456-345;
int u = a+20*35+234*456-345;
a = b *2 *c;
a = 436+235+23+20*35+234*456-345;
b = 400+234;
int k = 234;
b = a*k;
a = a*k+2000;
k = 234+a*c;
// Asignaciones de un solo numero
b = -1;
xxxx = 24;
k = 2000;
// Variables booleanas
boolean bandera = false;
boolean ban = false;
boolean ban1 = false;
boolean ban2 = false;
// Condición if-else
if(a < c){
    x = a+ b;
    a = 35+234*456-345;
} else {
    x = b - a * 100;
    b = a*k;
    a = a*k+2000;
}
x = 180;
// Ciclo While
while (x > 200) {
    k = x +2;
    if (k > x){
        k = 190;
    } else {
        k = k + 2 * x;
    }
    x = x - 1;
}
// Condición if
if (a < b && c < b){
    int aaa=9*5-32/3+42-(12*4)+5-32+54;
    int sss=93+243+42*993-43-4+32+32-12+3;
    int ccc=a12+a23*var4;
    int vvvv=a12-43+42+2+42-43;
```

```
int ttt=100 +31-32+32+32-31+53+432+321+984+90+443;
int jjj=102*32-31-32/32-21+32;
int hhh= a23*3-32+31+31-42+413-43*43;
int uuu=98*90*190*1902*9;
aaa = 324+23+235;
uuu = aaa * 2;
jjj = aaa * uuu;
jjj = jjj * 2;
a = 324+23+235;
b = aaa * 2;
c = aaa * uuu;
k = jjj * 2;
}

}
// Fin del metodo 1
}
// Fin de la clase
// ***** FIN DEL CODIGO FUENTE *****
```

Fuentes de referencia

- Gajardo, L., & Mateu, L. (2004). Análisis Semántico de programas escritos en Java. *Theoria*, 13(1), 39-49.
- Lezcano Matías, L., Sánchez Alonso, S., & Bravo-Agapito, J. (2019). Autómatas y procesadores de lenguajes.
- Zamora, F. R., & Varela, J. C. Generación de Código Intermedio.
- Jurado Málaga, E. (2008). *Teoría de autómatas y lenguajes formales*. Universidad de Extremadura. Servicio de Publicaciones.
- García José. *La optimización: Una mejora en la ejecución de programas*. Libro digital, recuperado de: <https://webs.um.es/jmgarcia/miwiki/lib/exe/fetch.php?media=ensayos.pdf>
- Hopcroft, J. E., Motwani, R., Ullman, J. D., & Alfonseca, M. (2002). Introducción a la teoría de autómatas, lenguajes y computación.
- Kelley, D., & Platas, M. L. D. (1995). *Teoría de autómatas y lenguajes formales* (Vol. 22). Prentice Hall.