UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

COLEGIO DE CIENCIA Y TECNOLOGÍA

LICENCIATURA EN INGENIERÍA DE SOFTWARE

Diseño y desarrollo de un software integrado (IDE, compilador y máquina virtual), para la enseñanza de la lógica de programación a partir de edades de 15 años, por medio de la creación propia de un lenguaje de programación de rutas instruccionales

T E S I S

QUE PARA OPTAR POR EL TÍTULO DE

LICENCIADO EN INGENIERÍA DE SOFTWARE

P R E S E N T A :

JONATHAN GARCÍA GIL

D I R E C T O R

MTRO. ADALBERTO ROBLES VALADEZ

Ciudad de México, mayo de 2023.

UNIVERSIDAD AUTÓNOMA DE LA CIUDAD DE MÉXICO

COLEGIO DE CIENCIA Y TECNOLOGÍA

ACADEMIA DE INFORMÁTICA

Licenciatura en Ingeniería de Software

"APÉNDICE 4: DESARROLLO DE LA MÁQUINA VIRTUAL"

TESIS

QUE PARA OPTAR POR EL TÍTULO DE LICENCIATURA EN:

INGENIERÍA DE SOFTWARE

PRESENTA:
JONATHAN GARCÍA GIL

DIRECTOR DE TESIS:
Mtro. Adalberto Robles Valadez
Profesor-Investigador de la Academia de Informática, UACM

Ciudad de México, mayo de 2023

Contenido

1 Máquina Virtual 1
1.1 Clase RunCode 1
1.1.1 Objetos que se obtienen de otras clases 1
1.1.2 Objetos Globales de la clase 2
1.1.3 Métodos para mostrar mensajes al usuario 3
1.1.4 Ejecutar la máquina virtual 3
1.1.5 Inicializando la máquina virtual 4
1.1.6 El método para copiar Mapas Virtuales 5
1.1.7 Obteniendo la ubicación del personaje 5
1.1.8 Declaración de funciones 5
1.1.9 El método para actualizar el mapa 6
1.1.10 Del mapa virtual al mapa visual 6
1.2 Resolver el programa del usuario 10
1.2.1 Generando número aleatorio 10
1.2.2 Resolviendo una operación aritmética 11
1.2.3 Funcionalidad: ¿Qué hay adelante de mí? 13
1.2.4 Declarando una variable 14
1.2.5 Modificar una variable 16
1.2.6 Incremento y decremento de variables 18
1.2.7 Imprimir variables 19
1.2.8 Imprimir una cadena compuesta 20
1.2.9 Función Avanzar 20
1.2.10 Método para pausar una instrucción 23
1.2.11 Función espera 23
1.2.12 Instrucción Pintar 25
1.2.13 Dejar de pintar 26
1.2.14 Girar a la izquierda 26
1.2.15 Girar a la derecha 28
1.2.16 Tomar objeto 29
1.2.17 Soltar objeto 30
1.2.18 Eliminar objeto 31

1.2.19 Desactivar una bomba 32
1.2.20 Terminar el programa 33
1.2.21 Evaluar condiciones 33
1.2.22 Funcionalidad: Resolviendo si existe una bomba, un objeto o muro delante del personaje 36
1.2.23 Estructura de control: (If-Then-Else) 36
1.2.24 Estructura de control: (Switch-Case) 38
1.2.25 Estructura de control: Do-While 40
1.2.26 Estructura de control: While 41
1.2.27 Estructura de control: For 41
1.2.28 Llamada a función 43
1.3 Ejecutar la máquina virtual 44
1.3.1 Método “isReady” 45
1.3.2 Limpiar el mapa sin ejecutar el programa del usuario 45
1.3.3 Ejecutar la máquina virtual 46
1.3.4 El método Run 46
1.3.5 Interrumpir el proceso de análisis 47

1 Máquina Virtual

Ahora ha llegado el turno del último componente importante del sistema: La máquina virtual. Esta parte del sistema será la encargada de leer el código JSON generado por el compilador, y ejecutará las instrucciones según se indiquen.

1.1 Clase RunCode

Esta clase es la máquina virtual, todos los elementos que se generan cuando se pasa el compilador y el mapa recaen en esta clase. Esta clase es el corazón del proyecto pues es la encargada de resolver el programa que el usuario escribió y mostrará los resultados de una forma gráfica. Es la última parte de este gran proyecto.

1.1.1 Objetos que se obtienen de otras clases

Este componente requiere de algunos objetos que se generan con anterioridad, las cuales son:

```
1.      private JLabel lblResultados;
2.      private JTextArea txtResultados;
3.      private JSONObject JSONPrograma;
4.      private int[][] MapaBackend;
5.      private int Filas;
6.      private int Columnas;
```

Dónde:

- lblResultados: es el objeto que se estará modificando continuamente con la finalidad de actualizar el mapa cada que se ejecute una instrucción. Este objeto tiene que ser el mismo que se genera en la clase *Fenix*.
- txtResultados: es el objeto en la que se mostrarán resultados cada que se ejecute una instrucción. Este objeto tiene que ser el mismo que se genera en la clase *Fenix*.
- JSONPrograma: es el objeto que contiene el JSON generado por el compilador.
- MapaBackend: es el mapa virtual que fue generado al momento de cargar el mapa del usuario.
- Filas: Son las filas máximas que tiene el Mapa Virtual.
- Columnas: Son las columnas máximas que tiene el Mapa Virtual.

Todos los elementos descritos anteriormente se obtienen de otras clases, por lo que se tiene que tener control de cuándo se generan y en qué momento se deben colocar en esta clase con sus debidos setters.

1.1.2 Objetos Globales de la clase

También existen objetos que se ocuparán de forma global, pero estos no son tomados de otras clases. Estos objetos son:

```
1.      private String MapaHTML;
2.      private int[][] MapaDeTrabajo;
3.      private int FilaPersonaje;
4.      private int ColumnaPersonaje;
5.      private int Personaje;
6.      private boolean tengoObjeto = false;
7.      private int DebajoDelPersonaje;
8.      private final cargarTexto idioma = new cargarTexto("RunCode");
9.      private TablaDeSimbolos tablaDeSimbolos = new TablaDeSimbolos();
10.     private int pintura = -1;
11.     private final Stack<TablaDeSimbolos> pilaDePilas = new Stack<>();
```

Dónde:

- MapaHTML: Es el mapa en forma de HTML que se coloca en el objeto lblResultados. Su estructura es parecida al que se ha definido al momento de cargar un mapa, pero cambia en algunos aspectos. De esto hablaré más adelante.
- MapaDeTrabajo: Se requiere de una copia completa del mapa virtual original con el objetivo de siempre tener los valores del mapa que se ha cargado. Si se trabajara con el objeto original (MapaBackend) no habrá forma de volver a obtener el mapa original si el usuario deseara ejecutar su programa una vez más.
- FilaPersonaje: Es un objeto de control, indica la fila actual dónde se encuentra el personaje cuando se está ejecutando el programa del usuario.
- ColumnaPersonaje: Es un objeto de control, indica la columna actual dónde se encuentra el personaje cuando se está ejecutando el programa del usuario.
- Personaje: Es la posición dónde está mirando el personaje, hay que recordar que tiene 4 posiciones diferentes las cuales son: Norte, Sur, Este u Oeste.
- tengoObjeto: Es un objeto de control. Indica que el personaje tiene un objeto del mapa, hay que recordar que no puede tener más de un objeto en posesión.
- DebajoDelPersonaje: Es un objeto de control. Guarda el elemento que está debajo del personaje, pues no siempre será una casilla en blanco. Hay que recordar que igualmente puede ser que el suelo haya sido pintado o una bandera.
- cargarTexto: Hay que recordar que es la clase que obtiene las cadenas necesarias en inglés o español.
- tablaDeSimbolos: Es la tabla de símbolos que se ha definido en el **Capítulo 8.6: Tabla de Símbolos del trabajo escrito**.
- pintura: En caso de haber ejecutado la instrucción “pintar()” este valor será el que contendrá el color indicado por el usuario.
- pilaDePilas: Es un objeto de control. Se utilizará principalmente cuando el usuario indique una llamada a función.

1.1.3 Métodos para mostrar mensajes al usuario

A lo largo de esta clase, se tendrán que mostrar diversos mensajes de error o de éxito. para ahorrar código se crearán 2 clases que escribirán en el objeto "*txtResultados*" el texto según sea el caso.

En caso de un mensaje normal se ocupará la clase "*agregarTexto*", esta clase recibe el texto que imprimirá en el objeto "*txtResultados*" y lo imprimirá de color verde.

```
1.      private void agregarTexto(String texto) {
2.          SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss ");
3.          Date date = new Date();
4.          this.txtResultados.setText(txtResultados.getText()  +  "\n"  +  formatter.format(date)  +
    texto);
5.          txtResultados.setForeground(new Color(25, 111, 61));
6.      }
```

En caso de error se ocupará el método "*agregarTextoError*" a diferencia del anterior, imprimirá el mensaje de color rojo.

```
1.      private void agregarTextoError(String texto) {
2.          SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss ");
3.          Date date = new Date();
4.          this.txtResultados.setText(txtResultados.getText()  +  "\n"  +  formatter.format(date)  +
    texto);
5.          this.txtResultados.setForeground(Color.red);
6.      }
```

En ambos casos antes del mensaje al principio, se escribe la hora exacta en la que se imprime el mensaje.

Si no se definen estos métodos el código se haría más complicado y en caso de cambiar los colores que se muestran al usuario, se tendrían que realizar más de 1 vez (Al final estos 2 métodos son utilizados 67 veces).

1.1.4 Ejecutar la máquina virtual

Para ejecutar la máquina virtual se tendrá que ejecutar el método "*ejecutaPrograma*", el algoritmo es el siguiente:

```
1.      public void ejecutaPrograma() {
2.          cleanMap();
3.          boolean resultado = this.resolverPrograma("main",JSONPrograma.getJSONArray("main"));
4.          if(resultado){
5.              agregarTexto(
6.                      idioma.traerTexto("ProgramaExitoso")
7.              );
8.          }else{
9.              agregarTextoError(
10.                     idioma.traerTexto("ProgramaFallido")
11.             );
12.         }
13.     }
14.
```

Lo primero que se debe realizar es preparar el programa para su ejecución, independientemente si se trata de una primera o enésima pasada. Siempre se tiene que inicializar los objetos anteriormente descritos. Esto se logra con el método "*cleanMap*();" y finalmente se ejecutan las instrucciones con el método "*resolverPrograma*".

El método "*resolverPrograma*" regresa un booleano que será verdadero o falso según sea el caso.

1.1.5 Inicializando la máquina virtual

El algoritmo para preparar las variables es el siguiente:

```
1.      public void cleanMap() {
2.          tablaDeSimbolos.limpiarTablaDeSimbolos();
3.          DebajoDelPersonaje = 0;
4.          MapaDeTrabajo = generarMapaDeTrabajo(MapaBackend);
5.          actualizarMapaPorMovimientos();
6.          tengoObjeto = false;
7.          actualizarMapa();
8.          verificaPersonaje();
9.          funcionesATablaDeSimbolos();
10.     }
```

Dónde:

- Primero se ejecuta el método *limpiarTablaDeSimbolos*() de la tabla de símbolos.
- Hay que recordar que al principio no se encuentra nada debajo del personaje, es decir, una bandera o que el tablero haya sido pintado. Por lo que *DebajoDelPersonaje* tiene que tener un valor 0.
- Como se dijo anteriormente, se tendrá que trabajar en un mapa virtual temporal para no afectar el mapa virtual original con el objetivo de siempre tenerlo disponible, ya que el mapa virtual no se vuelve a generar una vez que se leyó el archivo. Este mapa dónde se va a trabajar se guarda en "*MapaDeTrabajo*" ejecutando el método "*generarMapaDeTrabajo*".
- Igualmente, no importa si es la primera o enésima vez que se ejecuta el código del usuario, al principio el personaje no tiene un objeto cargando, por lo que este al iniciar siempre debe ser falso y se guarda el valor en "*tengoObjeto*".
- Una vez que se ha generado el mapa virtual dónde se va a trabajar, se tiene que actualizar el mapa visual, este paso está de sobra cuando se ejecuta por primera vez el programa del usuario, pero no cuando se ejecuta por enésima vez. Esto se logra con el método "*actualizarMapa*()".
- Después se tiene que verificar dónde se encuentra el personaje en el mapa virtual, con el objetivo de siempre tener presente su ubicación en el mapa. Esto se logra ejecutando el método "verificaPersonaje()".
- Finalmente, se tienen que declarar las funciones en la tabla de símbolos, para que cuando llegue el momento de una llamada a función, se pueda verificar que se han declarado las funciones y cuántos parámetros de entrada lleva esa función.

1.1.6 El método para copiar Mapas Virtuales

El algoritmo para copiar de un Mapa Virtual original a uno dónde se pueda trabajar es el siguiente:

```
1.      private int[][] generarMapaDeTrabajo(int[][] mapaBackend) {
2.          int[][] nuevoMapa = new int[Filas][Columnas];
3.          for(int i=0; i<Filas;i++){
4.              if (Columnas >= 0) System.arraycopy(mapaBackend[i], 0, nuevoMapa[i], 0, Columnas);
5.          }
6.          return nuevoMapa;
7.      }
```

1.1.7 Obteniendo la ubicación del personaje

Como se ha indicado, es necesario siempre tener presente la posición dónde se encuentra el personaje. Para obtenerlo, se utiliza el siguiente algoritmo que busca los números 1, 2, 3 o 4 en el mapa virtual:

```
1.      private void verificaPersonaje() {
2.          for (int i = 0; i < Filas; i++) {
3.              for (int j = 0; j < Columnas; j++) {
4.                  if ((MapaBackend[i][j] == 1) || (MapaBackend[i][j] == 2) || (MapaBackend[i][j] ==
    3) || (MapaBackend[i][j] == 4)) {
5.                      FilaPersonaje = i;
6.                      ColumnaPersonaje = j;
7.                      Personaje = MapaBackend[i][j];
8.                      return;
9.                  }
10.             }
11.         }
12.         agregarTextoError(idioma.traerTexto( "PersonajeNoEcontrado"));
13.     }
```

1.1.8 Declaración de funciones

En caso de que el usuario haya declarado funciones en su programa fuente, estos deben ser leídos antes de ejecutar las instrucciones, por lo tanto, se utiliza el siguiente algoritmo:

```
1.      private void funcionesATablaDeSimbolos() {
2.          JSONObject temporal;
3.          ArrayList temporal2;
4.          for(int i=0;i<JSONPrograma.getJSONArray("functions").length();i++){
5.              temporal = JSONPrograma.getJSONArray("functions").getJSONObject(i);
6.              temporal2 = (ArrayList) temporal.get("parameter");
7.              tablaDeSimbolos.agregarElementoATabla(
8.                      temporal.getString("name"),
9.                      temporal2.size(),
10.                     "funciones",
11.                     1
12.             );
13.         }
14.     }
```

1.1.9 El método para actualizar el mapa

Cada que ocurre un movimiento en el mapa se tiene que actualizar, además de actualizarse se debe esperar un tiempo que se puede configurar. El algoritmo es el siguiente:

```
1.      private void actualizarMapa() {
2.          generarMapaHTML();
3.          try {
4.              Thread.sleep(idioma.getConfigInteger("TiempoInstruccion"));
5.          } catch (InterruptedException e) {
6.              this.lblResultados.setText(MapaHTML);
7.          }
8.          this.lblResultados.setText(MapaHTML);
9.      }
```

1.1.10 Del mapa virtual al mapa visual

Anteriormente, para cargar un mapa se ha creado un algoritmo con hasta 8 casos diferentes. Sin embargo, para esta parte del programa se deben tomar en cuenta hasta 30 casos distintos los cuales son:

9. El suelo está pintado de color rojo sin el personaje.
10. El suelo está pintado de color azul sin el personaje.
11. El suelo está pintado de color verde sin el personaje.
12. El suelo está pintado de color amarillo sin el personaje.
13. Indica el lugar dónde el personaje ha muerto (En caso de haber tocado una bomba).
14. El suelo está pintado de color por defecto (naranja) sin el personaje.
15. Se trata de un objeto que se puede recoger, pero ha sido puesto sobre el suelo pintado de color rojo.
16. Se trata de un objeto que se puede recoger, pero ha sido puesto sobre el suelo pintado de color azul.
17. Se trata de un objeto que se puede recoger, pero ha sido puesto sobre el suelo pintado de color verde.
18. Se trata de un objeto que se puede recoger, pero ha sido puesto sobre el suelo pintado de color amarillo.
19. Se trata de un objeto que se puede recoger, pero ha sido puesto sobre el suelo pintado del color por defecto (naranja).
20. Se trata de la animación de un objeto eliminado sin color de fondo.
21. Se trata de la animación de un objeto eliminado con color de fondo rojo.
22. Se trata de la animación de un objeto eliminado con color de fondo azul.
23. Se trata de la animación de un objeto eliminado con color de fondo verde.
24. Se trata de la animación de un objeto eliminado con color de fondo amarillo.
25. Se trata de la animación de un objeto eliminado con color de fondo por default (naranja).
26. Se trata del símbolo de meta, pero el suelo ha sido pintado de color rojo.
27. Se trata del símbolo de meta, pero el suelo ha sido pintado de color azul.
28. Se trata del símbolo de meta, pero el suelo ha sido pintado de color verde.
29. Se trata del símbolo de meta, pero el suelo ha sido pintado de color amarillo.

30. Se trata del símbolo de meta, pero el suelo ha sido pintado de color por default (naranja).

Tomando en cuenta lo anterior, se ha modificado el algoritmo que anteriormente se planteó al momento de cargar el mapa visual, y se ha agregado el resto de casos que podrá tener.

```
1.      private void generarMapaHTML() {
2.          MapaHTML="";
3.          for (int i = 0; i < Filas; i++) {
4.              MapaHTML += "<tr>";
5.              for (int j = 0; j < Columnas; j++) {
6.                  switch (this.MapaDeTrabajo[i][j]) {
7.                      case 0:
8.                          MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("DEFAULT") + ";\">□</td>";
9.                          break;
10.                     case 1:
11.                         if (DebajoDelPersonaje == 9) {
12.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">⬆</td>";
13.                         } else if (DebajoDelPersonaje == 10) {
14.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">⬆</td>";
15.                         } else if (DebajoDelPersonaje == 11) {
16.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">⬆</td>";
17.                         } else if (DebajoDelPersonaje == 12) {
18.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">⬆</td>";
19.                         } else {
20.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("DEFAULT") + ";\">⬆</td>";
21.                         }
22.                         break;
23.                     case 2:
24.                         if (DebajoDelPersonaje == 9) {
25.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">⬇</td>";
26.                         } else if (DebajoDelPersonaje == 10) {
27.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">⬇</td>";
28.                         } else if (DebajoDelPersonaje == 11) {
29.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">⬇</td>";
30.                         } else if (DebajoDelPersonaje == 12) {
31.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">⬇</td>";
32.                         } else {
33.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("DEFAULT") + ";\">⬇</td>";
34.                         }
35.                         break;
36.                     case 3:
37.                         if (DebajoDelPersonaje == 9) {
38.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">➡</td>";
39.                         } else if (DebajoDelPersonaje == 10) {
40.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">➡</td>";
```

```
41.                         } else if (DebajoDelPersonaje == 11) {
42.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">➡</td>";
43.                         } else if (DebajoDelPersonaje == 12) {
44.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">➡</td>";
45.                         } else {
46.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("DEFAULT") + ";\">➡</td>";
47.                         }
48.                         break;
49.                     case 4:
50.                         if (DebajoDelPersonaje == 9) {
51.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">⬅</td>";
52.                         } else if (DebajoDelPersonaje == 10) {
53.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">⬅</td>";
54.                         } else if (DebajoDelPersonaje == 11) {
55.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">⬅</td>";
56.                         } else if (DebajoDelPersonaje == 12) {
57.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">⬅</td>";
58.                         } else {
59.                             MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("DEFAULT") + ";\">⬅</td>";
60.                         }
61.                         break;
62.                     case 5:
63.                         MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("DEFAULT") + ";\">🏁</td>";
64.                         break;
65.                     case 6:
66.                         MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("DEFAULT") + ";\">💣</td>";
67.                         break;
68.                     case 7:
69.                         MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("DEFAULT") + ";\">🚪</td>";
70.                         break;
71.                     case 8:
72.                         MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("DEFAULT") + ";\">❌</td>";
73.                         break;
74.                     case 9:
75.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("RED") +
    ";\">▓</td>";
76.                         break;
77.                     case 10:
78.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("BLUE") +
    ";\">▒</td>";
79.                         break;
80.                     case 11:
81.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("GREEN")
    + ";\">▒</td>";
82.                         break;
83.                     case 12:
84.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("YELLOW")
    + ";\">▒</td>";
85.                         break;
86.                     case 13:
```

```
87.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("RED") +
    ";\">🔥</td>";
88.                         break;
89.                     case 14:
90.                         MapaHTML        +=        "<td        style=\"background-color:"        +
    idioma.traerColor("COLORDEFAULT") + ";\">▨</td>";
91.                         break;
92.                     case 15:
93.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("RED") +
    ";\">🂠</td>";
94.                         break;
95.                     case 16:
96.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("BLUE") +
    ";\">🂠</td>";
97.                         break;
98.                     case 17:
99.                         MapaHTML += "<td style=\"background-color:" + idioma.traerColor("GREEN")
    + ";\">🂠</td>";
100.                            break;
101.                        case 18:
102.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">🂠</td>";
103.                            break;
104.                        case 19:
105.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("COLORDEFAULT") + ";\">🂠</td>";
106.                            break;
107.                        case 20:
108.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("DEFAULT") + ";\">✨</td>";
109.                            break;
110.                        case 21:
111.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">✨</td>";
112.                            break;
113.                        case 22:
114.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">✨</td>";
115.                            break;
116.                        case 23:
117.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">✨</td>";
118.                            break;
119.                        case 24:
120.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">✨</td>";
121.                            break;
122.                        case 25:
123.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("COLORDEFAULT") + ";\">✨</td>";
124.                            break;
125.                        case 26:
126.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("RED") + ";\">🏴</td>";
127.                            break;
128.                        case 27:
129.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("BLUE") + ";\">🏴</td>";
130.                            break;
131.                        case 28:
132.                            MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("GREEN") + ";\">🏴</td>";
```

```
                                break;
                            case 29:
                                MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("YELLOW") + ";\">🏁</td>";
                                break;
                            case 30:
                                MapaHTML       +=       "<td       style=\"background-color:"       +
    idioma.traerColor("COLORDEFAULT") + ";\">🏁</td>";
                                break;
                            default:

                        }
                    }
                    MapaHTML += "</tr>";
                }
                MapaHTML = "<html><body><br><table>" + MapaHTML + "</table><br></body></html>";
            }
```

1.2 Resolver el programa del usuario

El paso siguiente es resolver las instrucciones una por una hasta que no existan más instrucciones por ejecutar o el usuario haya ejecutado una función para terminar el programa. Para esto se tiene el siguiente algoritmo que se encarga de analizar cada una de las instrucciones generadas hasta terminar el programa.

```
    private boolean resolverPrograma(String funcionActual, JSONArray subprogram) {
        JSONObject temporal;
        for (int i = 0; i < subprogram.length(); i++) {
            temporal = subprogram.getJSONObject(i);
            switch (temporal.getString("instruction")) {
                case "«Palabra Clave»":
                    if (!«Llamada al método según sea el caso»()) {
                        return false;
                    }
                    break;
                default:
                    return false;
            }
        }
        return true;
    }
```

Todas las funcionalidades regresan un valor positivo o negativo. Positivo si la instrucción se ejecutó adecuadamente y negativo si existieron errores en la ejecución.

Cuando regresa un valor negativo el análisis se debe detener e indicar al usuario dónde ocurrió un error y por qué.

1.2.1 Generando número aleatorio

Una de las funciones que se utilizarán a menudo en las próximas instrucciones es la generación de números aleatorios. Aquí es dónde se resolverá esa cuestión. Si el usuario no indicó un rango de valores para generar un número aleatorio, este se generará entre el 1 y el 100. El algoritmo es el siguiente:

```
1.      private int generarAleatorio(int MIN, int MAX) {
2.          if (MIN == -1 && MAX == -1) {
3.              //Regresa el valor entre 1 y 100
4.              return (int) (Math.floor(Math.random() * (100 - 1 + 1)) + 1);
5.          } else {
6.              return (int) (Math.floor(Math.random() * (MAX - MIN + 1)) + MIN);
7.          }
8.      }
```

1.2.2 Resolviendo una operación aritmética

La operación aritmética no ha sido resuelta desde el compilador. El objetivo de este algoritmo es convertir la operación aritmética en una cadena de texto para utilizar un método que ayude a resolver una operación matemática.

Una operación aritmética puede ser compuesta de un número entero, el valor de una variable ya declarada, o de un número aleatorio. Igualmente se debe tener en cuenta los paréntesis que se abren o cierran en la operación aritmética.

Resolver un número entero es lo más sencillo, pues solo se coloca en la cadena de texto que se está formando. Para buscar el valor de una variable declarada, es necesario buscar ese valor en la tabla de símbolos. Y finalmente para generar un número aleatorio se ocupa el método anterior.

Para transformar e ir resolviendo cada una de estas cadenas se requiere el siguiente algoritmo:

```
1.      private String getStringOperacion(JSONObject operation) {
2.          String OperacionARealizar = "";
3.          JSONObject tmp;
4.
5.          switch (operation.getInt("type")) {
6.              case 0:
7.                  //Usuario abrió un paréntesis
8.                  if (Objects.equals(operation.getString("operator"), "")) {
9.                      OperacionARealizar += operation.getString("operator") + "(";
10.                     if (!Objects.equals(operation.getJSONObject("operation").toString(), "{}")) {
11.                         OperacionARealizar                                                    +=
    getStringOperacion(operation.getJSONObject("operation"));
12.                     }
13.                     OperacionARealizar += ")";
14.                 } else {
15.                     OperacionARealizar += operation.getString("operator");
16.                     if (!Objects.equals(operation.getJSONObject("operation").toString(), "{}")) {
17.                         OperacionARealizar                                                    +=
    getStringOperacion(operation.getJSONObject("operation"));
18.                     }
19.                 }
20.                 if (!Objects.equals(operation.getJSONObject("random").toString(), "{}")) {
21.                     OperacionARealizar += getStringOperacion(operation.getJSONObject("random"));
22.                 }
23.                 break;
24.             case 1:
25.                 //Es un número simple
26.                 OperacionARealizar += operation.getString("operator");
27.                 OperacionARealizar += operation.getInt("value");
28.                 if (!Objects.equals(operation.getJSONObject("operation").toString(), "{}")) {
```

```
29.                     OperacionARealizar                                                  +=
    getStringOperacion(operation.getJSONObject("operation"));
30.                 }
31.                 break;
32.             case 2:
33.                 //Es de un valor de una variable
34.                 //Verificando que exista la variable:
35.                 if (!tablaDeSimbolos.existeUnElemento(operation.getString("valueFrom"),  0)) {
36.                     agregarTextoError(
37.                             idioma.traerTexto( "ErrorAlTraerValorOperacion") +
38.                                     operation.getString("valueFrom")
39.                     );
40.                     OperacionARealizar += "ERROR";
41.                 }
42.                 OperacionARealizar += operation.getString("operator");
43.                 OperacionARealizar                                                      +=
    tablaDeSimbolos.valorDeUnIdentificador(operation.getString("valueFrom"),  0);
44.                 if(tablaDeSimbolos.valorDeUnIdentificador(operation.getString("valueFrom"),0)==-
    999999999){
45.                     agregarTextoError(
46.                             idioma.traerTexto("NoSeEncontroValorID")
47.                                     .replace("VAR",operation.getString("valueFrom"))
48.                     );
49.                     return "ERROR";
50.                 }
51.                 if (!Objects.equals(operation.getJSONObject("operation").toString(), "{}")) {
52.                     OperacionARealizar                                                  +=
    getStringOperacion(operation.getJSONObject("operation"));
53.                 }
54.                 break;
55.             case 3:
56.                 //Numero aleatorio
57.                 tmp = operation.getJSONObject("random");
58.                 OperacionARealizar += operation.getString("operator");
59.                 OperacionARealizar += generarAleatorio(tmp.getInt("from"), tmp.getInt("to"));
60.                 if (!Objects.equals(operation.getJSONObject("operation").toString(), "{}")) {
61.                     OperacionARealizar                                                  +=
    getStringOperacion(operation.getJSONObject("operation"));
62.                 }
63.                 break;
64.         }
65.         return OperacionARealizar;
66.     }
```

El algoritmo anterior puede pasar de la siguiente operación que declaró el usuario en su programa de entrada:

25+(id*aleatorio(1,7))

en algo como esto:

25+(4*5)

Si llegase a ocurrir un error, la cadena contendrá la cadena “ERROR” lo que imposibilitará la resolución más adelante. Esto ocurre principalmente si el valor se obtiene de una variable declarada y ésta no ha sido declarada aún. La cadena con el error tendrá la siguiente forma:

25+(ERROR*5)

Una vez que este algoritmo se ha ejecutado, se resolverá la operación matemática. Este algoritmo recibirá la cadena de texto lista para evaluarse, la evalúa y se guarda el valor en un objeto de tipo *float*.

Debido a que solo interesan solo números enteros (porque no tiene sentido moverse 1.598 bloques), se ocupará un método que convierta el número flotante en el número entero más cercano a éste.

Así es cómo se resuelve una operación aritmética (En caso de requerirlo), este es el algoritmo final:

```
1.      private int resolverOperacionAritmetica(JSONObject operation) {
2.          String OperacionAEvaluar = getStringOperacion(operation);
3.          ScriptEngineManager mgr = new ScriptEngineManager();
4.          ScriptEngine textoAOperacion = mgr.getEngineByName("JavaScript");
5.          if (!OperacionAEvaluar.contains("ERROR")) {
6.              try {
7.                  float                                   flotante                                =
    Float.parseFloat(textoAOperacion.eval(OperacionAEvaluar).toString());
8.                  return Math.round(flotante);
9.              } catch (Exception exe) {
10.                 return -999999999;
11.             }
12.         }
13.         return -999999999;
14.     }
```

Como paso a tener en cuenta: Cada que existe un error, se regresa un valor exageradamente negativo. Este error debe ser considerado por el algoritmo que llama a resolver una operación.

1.2.3 Funcionalidad: ¿Qué hay adelante de mí?

Esta funcionalidad está pensada para regresar un valor numérico que significa lo que tiene adelante el personaje. Es decir, regresará los siguientes valores dependiendo qué es lo que tiene en frente de él:

- Regresa el valor 0 si: No existe ningún obstáculo. Si el suelo está pintado de color se ignora y se considera como ningún obstáculo.
- Regresa el valor 1 si: Existe un objeto fijo. Hay que recordar que para este objeto el personaje no puede interactuar de ninguna manera.
- Regresa el valor 2 si: Es un objeto que puede tomar el personaje.
- Regresa el valor 3 si: El objeto es una bomba.
- Regresa el valor 4 si: El personaje se encuentra a los extremos del mapa.
- Regresa el valor 5 si: El objeto es la meta.
- Si llegase a existir algún error, esta función regresa el valor -100.

El algoritmo es el siguiente:

```
1.      private int resolverQueHayFrente() {
2.          //Comprobando que no sea tope de mapa
3.          if(FilaEnFrente()==FilaPersonaje && ColumnaEnFrente()==ColumnaPersonaje){
4.              //Esta al tope del mapa (No puede avanzar)
5.              return 4;
6.          }
7.          switch(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]){
8.              case 0: case 9: case 10: case 11: case 12: case 14:
9.                  //No hay obstáculo
10.                 return 0;
11.             case 5: case 26: case 27: case 28: case 29: case 30:
12.                 //El objeto es la meta
13.                 return 5;
14.             case 6:
15.                 //El objeto es una bomba
16.                 return 3;
17.             case 7: case 15: case 16: case 17: case 18: case 19:
18.                 //El objeto es un objeto que se puede mover
19.                 return 2;
20.             case 8:
21.                 //El objeto fijo
22.                 return 1;
23.         }
24.         return -100;
25.     }
```

1.2.4 Declarando una variable

La primera función del programa que analizaré será la declaración de una variable. Hay que recordar que, al momento de declarar una variable se puede asignar un valor. Este puede ser un valor numérico, un número aleatorio, una operación aritmética, de otra variable, declararse “en vacío” e incluso podría tener el valor de otra funcionalidad como es “quetengodelante()”.

Lo primero que realiza el algoritmo es comprobar en la tabla de símbolos que no haya sido declarado, de lo contrario tendrá que indicarle al usuario el error y detener la ejecución del análisis del programa (ninguna otra instrucción deberá ser ejecutada).

Después obtiene el valor numérico según sea el caso:

Si la variable se declara “en vacío” el valor por default será 0. En cualquier otro caso, la variable obtendrá su valor según sea el caso. El algoritmo es el siguiente:

```
1.      private boolean DECLARAVARIABLE(JSONObject temporal,String funcion) {
2.          if (tablaDeSimbolos.existeUnElemento(temporal.getString("identifier"),  0)) {
3.              agregarTexto(
4.                      idioma.traerTexto("InstruccionDECLARAVARIABLE")+
5.                              idioma.traerTexto("Ejecutada")
6.              );
7.              agregarTextoError(
8.                      idioma.traerTexto( "LaVariableExiste") +
9.                              temporal.getString("identifier") +
10.                             idioma.traerTexto( "EstaDeclarada")
11.             );
12.             return false;
13.         }
14.         switch (temporal.getInt("type")) {
15.             case 0:
16.                 //No hay valor de asignación
```

```
17.                 tablaDeSimbolos.agregarElementoATabla(
18.                         temporal.getString("identifier"),
19.                         0,
20.                         funcion,
21.                         0
22.                 );
23.                 break;
24.             case 1:
25.                 //Valor numérico
26.                 tablaDeSimbolos.agregarElementoATabla(
27.                         temporal.getString("identifier"),
28.                         temporal.getInt("value"),
29.                         funcion,
30.                         0
31.                 );
32.                 break;
33.             case 2:
34.                 //Número Aleatorio
35.                 JSONObject aleatorio = temporal.getJSONObject("random");
36.                 tablaDeSimbolos.agregarElementoATabla(
37.                         temporal.getString("identifier"),
38.                         generarAleatorio(aleatorio.getInt("from"), aleatorio.getInt("to")),
39.                         funcion,
40.                         0
41.                 );
42.                 break;
43.             case 3:
44.                 //Operación Aritmética
45.                 int                                     resultado                                       =
    this.resolverOperacionAritmetica(temporal.getJSONObject("operation"));
46.                 if(resultado==-999999999){
47.                     agregarTextoError(
48.                             idioma.traerTexto("ErrorOperacionAritmetica")
49.                     );
50.                     return false;
51.                 }
52.                 tablaDeSimbolos.agregarElementoATabla(
53.                         temporal.getString("identifier"),
54.                         resultado,
55.                         funcion,
56.                         0
57.                 );
58.                 break;
59.             case 4:
60.                 //De otra variable.
61.                 if (!tablaDeSimbolos.existeUnElemento(temporal.getString("valueFrom"),  0)) {
62.                     agregarTextoError(
63.                             idioma.traerTexto( "ErrorAlTraerValor") +
64.                                     temporal.getString("valueFrom") +
65.                                     idioma.traerTexto( "AsginarAlValor") +
66.                                     temporal.getString("identifier")
67.                     );
68.                     return false;
69.                 }
70.                 tablaDeSimbolos.agregarElementoATabla(
71.                         temporal.getString("identifier"),
72.                         tablaDeSimbolos.valorDeUnIdentificador(temporal.getString("valueFrom"),
    0),
73.                         funcion,
74.                         0
75.                 );
76.                 break;
77.             case 5:
78.                 //El valor proviene de una función QUETENGODELANTE()
79.                 tablaDeSimbolos.agregarElementoATabla(
```

```
                        temporal.getString("identifier"),
                        resolverQueHayFrente(),
                        funcion,
                        0
                );
                break;
        }
        return true;
    }
```

1.2.5 Modificar una variable

La siguiente funcionalidad es modificar el valor de una variable. Las acciones para esta parte son idénticas con excepción de que ahora se modificará la tabla de símbolos (no creará nuevos elementos).

El algoritmo es el siguiente:

```
1.      private boolean MODIFICAVARIABLE(JSONObject temporal) {
2.          String texto =
3.                  idioma.traerTexto("InstruccionMODIFICAVARIABLE")+
4.                          idioma.traerTexto("Ejecutada")
5.                  ;
6.          //Verifica si existe:
7.          if (!tablaDeSimbolos.existeUnElemento(
8.                  temporal.getString("identifier"),
9.                  0)) {
10.             agregarTexto(texto);
11.             agregarTextoError(
12.                     idioma.traerTexto( "LaVariableExiste") +
13.                             temporal.getString("identifier") +
14.                             idioma.traerTexto( "NoEstaDeclarada")
15.             );
16.             return false;
17.         }
18.
19.         String identificador = temporal.getString("identifier");
20.         temporal = temporal.getJSONObject("valueFrom");
21.
22.         switch (temporal.getInt("type")) {
23.             case 1:
24.                 //Valor numérico
25.                 agregarTexto(texto);
26.                 return tablaDeSimbolos.modificarElementoATabla(
27.                         identificador,
28.                         temporal.getInt("value"),
29.                         0
30.                 );
31.             case 2:
32.                 //Número Aleatorio
33.                 JSONObject aleatorio = temporal.getJSONObject("random");
34.                 agregarTexto(texto);
35.                 return tablaDeSimbolos.modificarElementoATabla(
36.                         identificador,
37.                         generarAleatorio(aleatorio.getInt("from"), aleatorio.getInt("to")),
38.                         0
39.                 );
40.             case 3:
41.                 //Operación Aritmética
42.                 int                                     resultado                                       =
    this.resolverOperacionAritmetica(temporal.getJSONObject("operation"));
43.                 if (resultado == -999999999) {
```

```
                    agregarTexto(texto);
                    agregarTextoError(
                            idioma.traerTexto("OperacionError")
                    );
                    return false;
                }
                agregarTexto(texto);
                return tablaDeSimbolos.modificarElementoATabla(
                        identificador,
                        resultado,
                        0
                );
            case 4:
                //De otra variable.
                if (!tablaDeSimbolos.existeUnElemento(temporal.getString("valueFrom"),  0)) {
                    agregarTextoError(
                            idioma.traerTexto( "ErrorAlTraerValor") +
                                    temporal.getString("valueFrom") +
                                    idioma.traerTexto( "AsginarAlValor") +
                                    identificador
                    );
                    agregarTexto(texto);
                    return false;
                }
                agregarTexto(texto);
                return tablaDeSimbolos.modificarElementoATabla(
                        identificador,
                        tablaDeSimbolos.valorDeUnIdentificador(temporal.getString("valueFrom"),
    0),
                        0
                );
            case 5:
                //El valor proviene de una función QUETENGODELANTE()
                agregarTexto(texto);
                return tablaDeSimbolos.modificarElementoATabla(
                        identificador,
                        resolverQueHayFrente(),
                        0
                );
        }
        agregarTexto(texto);
        return true;
    }
```

1.2.6 Incremento y decremento de variables

En muchas ocasiones es necesario realizar un incremento de variables de únicamente 1 valor. Es decir, si se tiene una variable con valor 7 y se requiere de aumentarla a 8. Para realizar lo anterior solo se consulta en la tabla de símbolos el valor actual de la variable a modificar y se incrementa o decrementa según sea el caso.

Los algoritmos para realizar lo anterior son los siguientes:

```
1.      private boolean DECREMENTAVARIABLE(JSONObject temporal) {
2.
3.          int repetir = tablaDeSimbolos.valorDeUnIdentificador(
4.                  temporal.getString("identifier"),
5.                  0
6.          );
7.          repetir-=1;
8.          agregarTexto(
9.                  temporal.getString("identifier")+
10.                         "-- " +
11.                         idioma.traerTexto("Ejecutada")
12.         );
13.         return tablaDeSimbolos.modificarElementoATabla(
14.                 temporal.getString("identifier"),
15.                 repetir,
16.                 0
17.         );
18.     }
19.
20.     private boolean INCREMENTAVARIABLE(JSONObject temporal) {
21.         int repetir = tablaDeSimbolos.valorDeUnIdentificador(
22.                 temporal.getString("identifier"),
23.                 0
24.         );
25.         repetir+=1;
26.         agregarTexto(
27.                 temporal.getString("identifier")+
28.                         "++ " +
29.                         idioma.traerTexto("Ejecutada")
30.         );
31.         return tablaDeSimbolos.modificarElementoATabla(
32.                 temporal.getString("identifier"),
33.                 repetir,
34.                 0
35.         );
36.     }
```

1.2.7 Imprimir variables

En muchos casos por temas de control, es necesario imprimir el valor de una variable. Para este propósito se ha creado esta función. El valor de una variable de momento se mostrará en el recuadro de resultados.

El algoritmo es el siguiente:

```
1.      private boolean IMPRIMIRVARIABLE(JSONObject temporal) {
2.          int valorDeLaVariable = tablaDeSimbolos.valorDeUnIdentificador(
3.                  temporal.getString("identifier"),
4.                  0
5.          );
6.          if(valorDeLaVariable==-999999999){
7.              agregarTextoError(
8.                      idioma.traerTexto("NoSeEncontroValorID")
9.                              .replace("VAR",temporal.getString("identifier"))
10.             );
11.             return false;
12.         }
13.         agregarTexto(
14.                 idioma.traerTexto("ElValorDeLaVariable")+
15.                         temporal.getString("identifier")+
16.                         idioma.traerTexto("EsElValor")+
17.                         valorDeLaVariable
18.         );
19.         return true;
20.     }
21.
```

1.2.8 Imprimir una cadena compuesta

Una alternativa al algoritmo anterior es este algoritmo. Es el único elemento que se ha traído de Java y consta de generar una cadena e imprimir variables según sea el caso con la estructura parecida a Java.

```
1.      private boolean IMPRIMIRCADENA(JSONObject temporal) {
2.          StringBuilder formarCadena = new StringBuilder();
3.          JSONObject temporal2;
4.          for(int i=0; i<temporal.getJSONArray("parameter").length();i++){
5.              temporal2 = temporal.getJSONArray("parameter").getJSONObject(i);
6.              switch (temporal2.getInt("type")){
7.                  case 0:
8.                      if(tablaDeSimbolos.valorDeUnIdentificador(temporal2.getString("valueFrom"),
    0)==-999999999){
9.                          agregarTextoError(
10.                                 idioma.traerTexto("NoSeEncontroValorID")
11.                                         .replace("VAR",temporal2.getString("valueFrom"))
12.                         );
13.                         return false;
14.                     }
15.                     formarCadena.append(tablaDeSimbolos.valorDeUnIdentificador(
16.                             temporal2.getString("valueFrom"),
17.                             0
18.                     ));
19.                 case 1:
20.                     formarCadena.append(temporal2.getString("string"));
21.             }
22.         }
23.         agregarTexto(formarCadena.toString());
24.         return true;
25.     }
26.
```

1.2.9 Función Avanzar

El primer algoritmo que se detallará será la acción de avanzar. Este algoritmo hace que el personaje avance por el tablero que el usuario ha cargado y mostrar resultados según corresponda.

Primero hay que definir un método que evalúe la casilla que está en frente del personaje y mostrar un aviso o ignorar según sea el caso. Adelante del personaje puede existir una casilla vacía que puede ocupar sin problemas, objetos fijos, bombas, objetos con los que puede interactuar, o banderas.

Hay que tomar en cuenta lo siguiente:

- Si hay una bomba debe terminar de ejecutar las instrucciones.
- Mostrar un mensaje de advertencia si se encuentra en un objeto que no puede atravesar, pero no termina la ejecución de instrucciones.
- Si existen banderas de meta en frente del personaje se debe guardar en una variable temporal, además siempre se debe considerar que si el personaje está pintando el suelo debe seguir pintando aún con el objeto.
- Se debe considerar si el personaje está pintando el suelo para avanzar.

El algoritmo que resuelve lo anterior es el siguiente:

```
1.      private boolean avanzarSimple(){
2.          //Sin valor, solo se mueve 1 vez
3.          if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==6){
4.              //Es bomba (termina el juego)
5.              MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
6.              MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 13;
7.              actualizarMapa();
8.              agregarTextoError(
9.                      idioma.traerTexto("PersonajeMuerto")
10.             );
11.             return false;
12.         }else             if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==8             ||
    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==7){
13.             //Cualquier cosa menos espacios en blanco y banderas. (No se debe mover)
14.             agregarTexto(
15.                     idioma.traerTexto("ObjetoNoTraspasable")
16.             );
17.         }else
    if(((MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==5)||(DebajoDelPersonaje==26)||
18.
    (MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==27)||(MapaDeTrabajo[FilaEnFrente()][ColumnaEn
    Frente()]==28)||
19.
    (MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==29)||(MapaDeTrabajo[FilaEnFrente()][ColumnaEn
    Frente()]==30))&&pintura!=-1){
20.             //Enfrente está una bandera.
21.             MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
22.             switch (pintura){
23.                 case 9:
24.                     DebajoDelPersonaje = 26;
25.                     break;
26.                 case 10:
27.                     DebajoDelPersonaje = 27;
28.                     break;
29.                 case 11:
30.                     DebajoDelPersonaje = 28;
31.                     break;
32.                 case 12:
33.                     DebajoDelPersonaje = 29;
34.                     break;
35.                 case 14:
36.                     DebajoDelPersonaje = 30;
37.                     break;
38.             }
39.             MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = Personaje;
40.             FilaPersonaje=FilaEnFrente();
41.             ColumnaPersonaje=ColumnaEnFrente();
42.         }else{
43.             MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
44.             if(pintura==-1){
45.                 DebajoDelPersonaje = MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()];
46.             }else{
47.                 DebajoDelPersonaje = pintura;
48.             }
49.             MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = Personaje;
50.             FilaPersonaje=FilaEnFrente();
51.             ColumnaPersonaje=ColumnaEnFrente();
52.         }
53.         return true;
54.     }
55.
```

Ahora, regresando a la función para avanzar, se debe considerar que adentro de los paréntesis de una instrucción “avanzar()” puede estar vacío (es decir, solo se moverá 1 vez), puede tener un número natural, una operación aritmética, o resolver un número aleatorio.

Antes de continuar quiero ser puntual en algo, si en el resultado de una operación aritmética éste llega a dar un número negativo, este se considerará un 0. Jamás se ejecutará la instrucción.

```
1.      private boolean avanzarSimple(){
2.          //Sin valor, solo se mueve 1 vez
3.          if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==6){
4.              //Es bomba (termina el juego)
5.              MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
6.              MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 13;
7.              actualizarMapa();
8.              agregarTextoError(
9.                      idioma.traerTexto("PersonajeMuerto")
10.             );
11.             return false;
12.         }else             if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==8             ||
    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==7){
13.             //Cualquier cosa menos espacios en blanco y banderas. (No se debe mover)
14.             agregarTexto(
15.                     idioma.traerTexto("ObjetoNoTraspasable")
16.             );
17.         }else
    if(((MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==5)||(DebajoDelPersonaje==26)||
18.
    (MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==27)||(MapaDeTrabajo[FilaEnFrente()][ColumnaEn
    Frente()]==28)||
19.
    (MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()]==29)||(MapaDeTrabajo[FilaEnFrente()][ColumnaEn
    Frente()]==30))&&pintura!=-1){
20.             //Enfrente está una bandera.
21.             MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
22.             switch (pintura){
23.                 case 9:
24.                     DebajoDelPersonaje = 26;
25.                     break;
26.                 case 10:
27.                     DebajoDelPersonaje = 27;
28.                     break;
29.                 case 11:
30.                     DebajoDelPersonaje = 28;
31.                     break;
32.                 case 12:
33.                     DebajoDelPersonaje = 29;
34.                     break;
35.                 case 14:
36.                     DebajoDelPersonaje = 30;
37.                     break;
38.             }
39.             MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = Personaje;
40.             FilaPersonaje=FilaEnFrente();
41.             ColumnaPersonaje=ColumnaEnFrente();
42.         }else{
43.             MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = DebajoDelPersonaje;
44.             if(pintura==-1){
45.                 DebajoDelPersonaje = MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()];
46.             }else{
```

```
47.                 DebajoDelPersonaje = pintura;
48.             }
49.             MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = Personaje;
50.             FilaPersonaje=FilaEnFrente();
51.             ColumnaPersonaje=ColumnaEnFrente();
52.         }
53.         return true;
54.     }
```

El último paso de este algoritmo es mostrar los resultados y actualizar el mapa.

1.2.10 Método para pausar una instrucción

Este método se ejecutará cada vez que se termine de ejecutar una instrucción. Lo que realiza es generar el mapa visual, esperar unos segundos (configurables) y actualizar el mapa visual.

Si no se realiza esta pausa cada que se termina una instrucción no se podrá observar paso a paso las instrucciones que realiza el personaje. Sin esto, pasará en menos de 1 segundo a mostrar el final del programa.

El algoritmo es el siguiente:

```
1.      private void actualizarMapaString(String texto) {
2.          generarMapaHTML();
3.          try {
4.              Thread.sleep(idioma.getConfigInteger("TiempoInstruccion"));
5.          } catch (InterruptedException e) {
6.              this.lblResultados.setText(MapaHTML);
7.          }
8.          agregarTexto(texto);
9.          this.lblResultados.setText(MapaHTML);
10.     }
```

1.2.11 Función espera

Esta función lo único que realiza es esperar n veces un tiempo configurable. Al igual que la instrucción avanzar() resuelve lo que se pone dentro del paréntesis en caso de existir y después ejecuta la instrucción espera() las veces que indicó el usuario.

```
1.      private boolean ESPERA(JSONObject temporal) {
2.          temporal = temporal.getJSONObject("parameter");
3.          int repetir;
4.
5.          switch (temporal.getInt("type")) {
6.              case 0:
7.                  agregarTexto(
8.                          idioma.traerTexto("Esperando")+
9.                                  "1"+
10.                                 idioma.traerTexto("Segundos")
11.                 );
12.                 esperaTiempo();
13.                 break;
14.             case 1:
15.                 //El valor viene de un identificador
16.                 repetir = tablaDeSimbolos.valorDeUnIdentificador(
17.                         temporal.getString("identifier"),
18.                         0
```

```
                );
                if(repetir==-999999999){
                    agregarTextoError(
                            idioma.traerTexto("NoSeEncontroValorID")
                                    .replace("VAR",temporal.getString("identifier"))
                    );
                    return false;
                }
                for(int i=0;i<repetir;i++){
                    agregarTexto(
                            idioma.traerTexto("Esperando")+
                                    repetir+
                                    idioma.traerTexto("Segundos")
                    );
                    esperaTiempo();
                }
                break;
            case 2:
                //El valor viene de un valor numérico
                repetir = temporal.getInt("value");
                agregarTexto(
                        idioma.traerTexto("Esperando")+
                                repetir+
                                idioma.traerTexto("Segundos")
                );
                for(int i=0;i<repetir;i++){
                    esperaTiempo();
                }
                break;
            case 3:
                //El valor viene de un número aleatorio
                temporal = temporal.getJSONObject("valueFrom");
                repetir = generarAleatorio(temporal.getInt("from"),temporal.getInt("to"));
                agregarTexto(
                        idioma.traerTexto("Esperando")+
                                repetir+
                                idioma.traerTexto("Segundos")
                );
                for(int i=0;i<repetir;i++){
                    esperaTiempo();
                }
                break;
            case 4:
                //El valor viene de una operación aritmética
                temporal = temporal.getJSONObject("valueFrom");
                repetir = resolverOperacionAritmetica(temporal);
                if(repetir==-999999999){
                    agregarTextoError(
                            idioma.traerTexto("ErrorOperacionAritmetica")
                    );
                    return false;
                }
                agregarTexto(
                        idioma.traerTexto("Esperando")+
                                repetir+
                                idioma.traerTexto("Segundos")
                );
                for(int i=0;i<repetir;i++){
                    esperaTiempo();
                }
                break;
        }
        return true;
    }
```

```
84.     private void esperaTiempo() {
85.         try {
86.             Thread.sleep(idioma.getConfigInteger("TiempoInstruccion"));
87.         } catch (InterruptedException e) {
88.             this.lblResultados.setText(MapaHTML);
89.         }
90.     }
```

1.2.12 Instrucción Pintar

La instrucción pintar será la encargada de colorear cada casilla mientras avanza el personaje.

El algoritmo para poder realizar esta función es sencillo, solo toma en cuenta que exista una casilla de meta pues es el único objeto dónde se puede sobreponer el personaje sin recogerlo.

El algoritmo es el siguiente:

```
1.      private boolean PINTAR(JSONObject temporal) {
2.          String texto =
3.                  idioma.traerTexto("InstruccionPINTAR")+
4.                          idioma.traerTexto("Ejecutada")
5.          ;
6.          boolean                                         haybandera                                          =
    (DebajoDelPersonaje==5)||(DebajoDelPersonaje==26)||(DebajoDelPersonaje==27)||
7.                  (DebajoDelPersonaje==28)||(DebajoDelPersonaje==29)||(DebajoDelPersonaje==30);
8.          switch (temporal.getString("color")){
9.              case "azul": case "blue":
10.                 if(haybandera){
11.                     pintura = 27;
12.                 }else{
13.                     pintura = 10;
14.                 }
15.                 break;
16.             case "red": case "rojo":
17.                 if(haybandera){
18.                     pintura = 26;
19.                 }else{
20.                     pintura = 9;
21.                 }
22.                 break;
23.             case "verde": case "green":
24.                 if(haybandera){
25.                     pintura = 28;
26.                 }else{
27.                     pintura = 11;
28.                 }
29.                 break;
30.             case "amarillo": case "yellow":
31.                 if(haybandera){
32.                     pintura = 29;
33.                 }else{
34.                     pintura = 12;
35.                 }
36.                 break;
37.             default:
38.                 if(haybandera){
39.                     pintura = 30;
40.                 }else{
41.                     pintura = 14;
```

```
                }
            }
            DebajoDelPersonaje = pintura;
            agregarTexto(texto);
            actualizarMapa();
            return true;
        }
```

1.2.13 Dejar de pintar

Para ejecutar la opción de dejar de pintar el suelo, se ocupa el siguiente algoritmo. Solo se modifica la variable global “pintura”.

```
    private boolean DEJAPINTAR() {
        pintura = -1;
        agregarTexto(
                idioma.traerTexto("InstruccionDejarPintar")+
                        idioma.traerTexto("Ejecutada")
        );
        return true;
    }
```

1.2.14 Girar a la izquierda

Para girar el personaje a la izquierda es necesario dirigir a el personaje en la dirección correcta. Si el personaje está en una posición, tendrá que girar en sentido contrario a las manecillas del reloj.

```
    private int getIzquierda(){
        switch (Personaje){
            case 1:
                return 4;
            case 2:
                return 3;
            case 3:
                return 1;
            case 4:
                return 2;
        }
        return -1;
    }
```

Después, se tendrá que resolver los parámetros de la función (en caso de existir). Estos pueden ser que no tenga parámetros (Se ejecutará la instrucción una sola vez), el valor de una variable, el valor numérico, un número aleatorio, o de una operación aritmética.

```
    private boolean IZQUIERDA(JSONObject temporal) {
        temporal = temporal.getJSONObject("parameter");
        int repetir;

        String texto =
                    idioma.traerTexto("InstruccionIzquierda")+
                            idioma.traerTexto("Ejecutada")
                ;

        switch (temporal.getInt("type")) {
            case 0:
                Personaje = getIzquierda();
```

```
13.                 MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
14.                 actualizarMapaString(texto);
15.                 break;
16.             case 1:
17.                 //El valor viene de un identificador
18.                 repetir = tablaDeSimbolos.valorDeUnIdentificador(
19.                         temporal.getString("identifier"),
20.                         0
21.                 );
22.                 if(repetir==-999999999){
23.                     agregarTextoError(
24.                             idioma.traerTexto("NoSeEncontroValorID")
25.                                     .replace("VAR",temporal.getString("identifier"))
26.                     );
27.                     return false;
28.                 }
29.                 for(int i=0;i<repetir;i++){
30.                     Personaje = getIzquierda();
31.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
32.                     actualizarMapaString(texto);
33.                 }
34.                 break;
35.             case 2:
36.                 //El valor viene de un valor numérico
37.                 repetir = temporal.getInt("value");
38.                 for(int i=0;i<repetir;i++){
39.                     Personaje = getIzquierda();
40.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
41.                     actualizarMapaString(texto);
42.                 }
43.                 break;
44.             case 3:
45.                 //El valor viene de un número aleatorio
46.                 temporal = temporal.getJSONObject("valueFrom");
47.                 repetir = generarAleatorio(temporal.getInt("from"),temporal.getInt("to"));
48.                 for(int i=0;i<repetir;i++){
49.                     Personaje = getIzquierda();
50.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
51.                     actualizarMapaString(texto);
52.                 }
53.                 break;
54.             case 4:
55.                 //El valor viene de una operación aritmética
56.                 temporal = temporal.getJSONObject("valueFrom");
57.                 repetir = resolverOperacionAritmetica(temporal);
58.                 if(repetir==-999999999){
59.                     agregarTextoError(
60.                             idioma.traerTexto("ErrorOperacionAritmetica")
61.                     );
62.                     return false;
63.                 }
64.                 for(int i=0;i<repetir;i++){
65.                     Personaje = getIzquierda();
66.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
67.                     actualizarMapaString(texto);
68.                 }
69.                 break;
70.         }
71.         return true;
72.     }
```

1.2.15 Girar a la derecha

Realiza las mismas acciones que el algoritmo anterior, con la diferencia de que el personaje debe girar en sentido de las manecillas del reloj.

```
1.      private int getDerecha(){
2.          switch (Personaje){
3.              case 1:
4.                  return 3;
5.              case 2:
6.                  return 4;
7.              case 3:
8.                  return 2;
9.              case 4:
10.                 return 1;
11.         }
12.         return -1;
13.     }
14.     private boolean DERECHA(JSONObject temporal) {
15.         temporal = temporal.getJSONObject("parameter");
16.         int repetir;
17.
18.         String texto =
19.                 idioma.traerTexto("InstruccionDerecha")+
20.                         idioma.traerTexto("Ejecutada")
21.                 ;
22.
23.         switch (temporal.getInt("type")) {
24.             case 0:
25.                 Personaje = getDerecha();
26.                 MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
27.                 actualizarMapaString(texto);
28.                 break;
29.             case 1:
30.                 //El valor viene de un identificador
31.                 repetir = tablaDeSimbolos.valorDeUnIdentificador(
32.                         temporal.getString("identifier"),
33.                         0
34.                 );
35.                 if(repetir==-999999999){
36.                     agregarTextoError(
37.                             idioma.traerTexto("NoSeEncontroValorID")
38.                                     .replace("VAR",temporal.getString("identifier"))
39.                     );
40.                     return false;
41.                 }
42.                 for(int i=0;i<repetir;i++){
43.                     Personaje = getDerecha();
44.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
45.                     actualizarMapaString(texto);
46.                 }
47.                 break;
48.             case 2:
49.                 //El valor viene de un valor numérico
50.                 repetir = temporal.getInt("value");
51.                 for(int i=0;i<repetir;i++){
52.                     Personaje = getDerecha();
53.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
54.                     actualizarMapaString(texto);
55.                 }
56.                 break;
57.             case 3:
58.                 //El valor viene de un número aleatorio
```

```
59.                 temporal = temporal.getJSONObject("valueFrom");
60.                 repetir = generarAleatorio(temporal.getInt("from"),temporal.getInt("to"));
61.                 for(int i=0;i<repetir;i++){
62.                     Personaje = getDerecha();
63.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
64.                     actualizarMapaString(texto);
65.                 }
66.                 break;
67.             case 4:
68.                 //El valor viene de una operación aritmética
69.                 temporal = temporal.getJSONObject("valueFrom");
70.                 repetir = resolverOperacionAritmetica(temporal);
71.                 if(repetir==-999999999){
72.                     agregarTextoError(
73.                             idioma.traerTexto("ErrorOperacionAritmetica")
74.                     );
75.                     return false;
76.                 }
77.                 for(int i=0;i<repetir;i++){
78.                     Personaje = getDerecha();
79.                     MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = Personaje;
80.                     actualizarMapaString(texto);
81.                 }
82.                 break;
83.         }
84.         return true;
85.     }
86.
```

1.2.16 Tomar objeto

Esta función se ejecuta si existe un objeto con el que el personaje pueda interactuar. Aunque, se toma en cuenta lo siguiente:

- El personaje no puede tomar más de un objeto a la vez.
- El personaje puede que no tenga algún objeto con el que pueda interactuar.
- Si existe una bomba en frente del personaje y la intenta tomar durante la ejecución, el programa termina.

El algoritmo que realiza esto es el siguiente:

```
1.      private boolean TOMAR() {
2.          String texto;
3.          if(tengoObjeto){
4.              //El personaje ya tiene un objeto, no puede tener 2 a la vez
5.              texto=
6.                      idioma.traerTexto("YaTengoObjeto")
7.              ;
8.          }else if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] == 6){
9.              //Toma una bomba, el juego termina
10.             MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = 13;
11.             MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 13;
12.             texto=
13.                     idioma.traerTexto("TomasteKaboom")
14.             ;
15.             agregarTextoError(texto);
16.             actualizarMapa();
17.             return false;
18.         }else if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] != 7){
19.             //No hay objeto que tomar
20.             texto=
```

```
                    idioma.traerTexto("NingunObjeto")
            ;
        }else{
            //Toma el objeto en frente del personaje
            tengoObjeto = true;
            MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 0 ;
            texto=
                    idioma.traerTexto("InstruccionObjeto")+
                            idioma.traerTexto("Ejecutada")
            ;
        }
        actualizarMapaString(texto);
        return true;
    }
```

1.2.17 Soltar objeto

Una vez que el personaje ha tomado un objeto, este puede soltarlo en dónde desee. Aunque se debe tomar en cuenta lo siguiente:

- Si el personaje intenta dejar un objeto en una bomba, la ejecución del programa termina.
- Puede dejar el objeto en un espacio vacío con suelo pintado, pero al soltar el personaje el objeto el suelo debe permanecer pintado.
- Solo puede dejar un objeto sobre un espacio vacío.

El algoritmo que se forma es el siguiente:

```
    private boolean SOLTAR() {
        String texto = "";
        if(!tengoObjeto){
            texto=
                    idioma.traerTexto("SinObjeto")
            ;
            actualizarMapaString(texto);
            return true;
        }
        if(tengoEspacioParaDejarObjeto()){
            int colorPintado = MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()];
            switch (colorPintado){
                case 0:
                    //No hay Nada en frente
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 7;
                    break;
                case 6:
                    //Kaboom
                    MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = 13;
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 13;
                    agregarTextoError(idioma.traerTexto("TocasteKaboom"));
                    actualizarMapa();
                    return false;
                case 9:
                    //No hay Nada en frente y el suelo es de color rojo
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 15;
                    break;
                case 10:
                    //No hay Nada en frente y el suelo es de color azul
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 16;
                    break;
                case 11:
                    //No hay Nada en frente y el suelo es de color verde
```

```
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 17;
                    break;
                case 12:
                    //No hay Nada en frente y el suelo es de color amarillo
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 18;
                    break;
                case 14:
                    //No hay Nada en frente y el suelo es de color orange
                    MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 19;
                    break;
                default:
                    //Hay una bandera (5,26-30), Un objeto (7,15-19), Un objeto inaccesible (8)
                    actualizarMapaString(idioma.traerTexto("SinEspacio"));
                    return true;
            }
            tengoObjeto = false;
            texto=
                    idioma.traerTexto("InstruccionSoltar")+
                            idioma.traerTexto("Ejecutada")
            ;
        }
        actualizarMapaString(texto);
        return true;
    }
```

1.2.18 Eliminar objeto

Es un objeto con el que puede interactuar el personaje y se puede eliminar. Lo único que se tiene que tener en consideración es que si intenta eliminar una bomba el análisis del programa termina y que si elimina un objeto con suelo pintado al eliminarlo deberá quedar de la misma forma.

El algoritmo es el siguiente:

```
    private boolean ELIMINAR() {
        String texto;
        int objetoFrente = MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()];
        switch (objetoFrente){
            case 6:
                MapaDeTrabajo[FilaPersonaje][ColumnaPersonaje] = 13;
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 13;
                agregarTextoError(idioma.traerTexto("TocasteKaboom"));
                actualizarMapa();
                return false;
            case 7:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 20 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 0 ;
                break;
            case 15:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 21 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 9 ;
                break;
            case 16:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 22 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 10 ;
```

```
                break;
            case 17:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 23 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 11 ;
                break;
            case 18:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 24 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 12 ;
                break;
            case 19:
                //Elimina el objeto en frente del personaje
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 25 ;
                actualizarMapa();
                MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 14 ;
                break;
            default:
                actualizarMapaString(idioma.traerTexto("NingunObjeto"));
        }
        texto=
                idioma.traerTexto("InstruccionEliminarObjeto")+
                        idioma.traerTexto("Ejecutada")
        ;
        actualizarMapaString(texto);
        return true;
    }
```

1.2.19 Desactivar una bomba

Hasta ahora se ha hablado de que el personaje puede “morir” si toca una bomba, o si intenta dejar o tomar un objeto sobre una casilla con una bomba. Afortunadamente estas bombas se pueden desactivar con la instrucción *DesactivarBomba*().

Cuando el personaje desactiva la bomba, ésta se convierte en un objeto con el que puede interactuar el personaje, o eliminarla del camino.

El algoritmo es el siguiente:

```
1.      private boolean DESACTIVARKABOOM() {
2.          String texto;
3.          if(MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] == 6){
4.              MapaDeTrabajo[FilaEnFrente()][ColumnaEnFrente()] = 7;
5.              texto=
6.                      idioma.traerTexto("InstruccionKaboomDesactivada")
7.              ;
8.          }else{
9.              texto=
10.                     idioma.traerTexto("NingunaBomba")
11.             ;
12.         }
13.         actualizarMapaString(texto);
14.         return true;
15.     }
16.
```

1.2.20 Terminar el programa

En el caso de que se intente terminar el programa antes de finalizar con el análisis de éste, se puede ejecutar la siguiente instrucción.

Lo único que realizará este algoritmo es informar al usuario que ha terminado el análisis de su programa porque se ejecutó esta instrucción.

```
1.      private boolean TERMINARBLOQUE() {
2.          agregarTexto(
3.                  idioma.traerTexto("TerminarBloqueUsuario")
4.          );
5.          agregarTexto(
6.                  idioma.traerTexto("TerminarBloque")+
7.                          idioma.traerTexto("Ejecutada")
8.          );
9.          return true;
10.     }
```

1.2.21 Evaluar condiciones

Antes de pasar a las estructuras de control es necesario definir cómo resuelve las condiciones la máquina virtual.

Hay que recordar que una evaluación de condiciones se compone de hasta 9 casos posibles las cuales son:

- El tipo es 0 si: El valor siempre es falso o verdadero.
- El tipo es 1 si: El valor a comparar proviene de un identificador.
- El tipo es 2 si: El valor a comparar es un valor numérico.
- El tipo es 3 si: El valor a comparar es un número aleatorio.
- El tipo es 4 si: La comparación está encerrada en un paréntesis. Hay que tener en cuenta si es tipo 4 puede existir la posibilidad que se tenga que negar la evaluación de la condición. Es decir, el usuario ingresó algo como: !(true).
- El tipo es 5 si: Se tiene que evaluar una condición lógica.
- El tipo es 6 si: Se ejecuta la función *tengo_muro_delante();* y puede regresar los valores verdadero o falso.
- El tipo es 7 si: Se ejecuta la función *tengo_objeto_delante();* y puede regresar los valores verdadero o falso.
- El tipo es 8 si: Se ejecuta la función *tengo_bomba_delante();* y puede regresar los valores verdadero o falso.
- El tipo es 9 si: Se ejecuta la función *delante_de_mi();* de cuyos valores de retorno se hablaron en el Capítulo 1.2.3 Funcionalidad: ¿Qué hay adelante de mí?

Se requieren evaluar los valores de la izquierda o derecha de una condición, es decir, realizar lo siguiente:

ValorDeLaIzquierda Comparador ValorDeLaDerecha

Por ejemplo: 7 < 9

Aunque no siempre se cumple un valor de la derecha, como puede ser:

true (siempre verdadero)

(se ejecuta la función tengo_objeto_delante() y regresa un valor falso)

Es la razón por la que en el algoritmo que aparecerá en breve el valor de la derecha solo considera los casos 1, 2, 3 y 9.

```
1.      private boolean evaluarCondicion(JSONObject condition) {
2.          int valorIzquierdo = 0;
3.          int valorDerecho = 0;
4.          String tipoOperacion = "";
5.          switch (condition.getInt("type")){
6.              case 0:
7.                  //Siempre falso o verdadero
8.                  switch (condition.getString("value")){
9.                      case "true": case "verdadero":
10.                         return true;
11.                     case "false": case "falso":
12.                         return false;
13.                 }
14.                 break;
15.             case 1:
16.                 //El valor proviene de un identificador
17.                 valorIzquierdo = tablaDeSimbolos.valorDeUnIdentificador(
18.                         condition.getString("identifier"),
19.                         0
20.                 );
21.                 if(valorIzquierdo==-999999999){
22.                     agregarTextoError(
23.                             idioma.traerTexto("NoSeEncontroValorID")
24.                                     .replace("VAR",condition.getString("identifier"))
25.                     );
26.                     try {
27.                         throw new Exception();
28.                     } catch (Exception e) {
29.                         throw new RuntimeException(e);
30.                     }
31.                 }
32.                 break;
33.             case 2:
34.                 //El valor proviene de un número
35.                 valorIzquierdo = condition.getInt("value");
36.                 break;
37.             case 3:
38.                 //El valor proviene de un número aleatorio
39.                 valorIzquierdo = generarAleatorio(
40.                         condition.getJSONObject("valueFrom").getInt("from"),
41.                         condition.getJSONObject("valueFrom").getInt("to")
42.                 );
43.                 break;
44.             case 4:
45.                 //comparación en paréntesis
46.                 if(condition.getBoolean("value")){
47.                     //Se tiene que negar al final.
48.                     return !evaluarCondicion(condition.getJSONObject("valueFrom"));
49.                 }else{
50.                     return evaluarCondicion(condition.getJSONObject("valueFrom"));
51.                 }
52.             case 5:
53.                 //comparación lógica
```

```
54.                 boolean izquierda = evaluarCondicion(condition.getJSONObject("valueFrom"));
55.                 boolean derecha = evaluarCondicion(condition.getJSONObject("compareWith"));
56.                 String OperadorLogico = condition.getString("identifier");
57.                 if(Objects.equals(OperadorLogico, "or")){
58.                     return (izquierda || derecha);
59.                 }else if(Objects.equals(OperadorLogico, "and")){
60.                     return (izquierda && derecha);
61.                 }
62.                 break;
63.             case 6:
64.                 //Pregunta si hay muro en frente
65.                 return resolverMuroFrente();
66.             case 7:
67.                 //Pregunta si hay muro en frente
68.                 return resolverObjetoFrente();
69.             case 8:
70.                 //Pregunta si hay muro en frente
71.                 return resolverKaboomFrente();
72.             case 9:
73.                 //Pregunta si hay muro en frente
74.                 valorIzquierdo = resolverQueHayFrente();
75.                 break;
76.         }
77.
78.         condition = condition.getJSONObject("compareWith");
79.         switch (condition.getInt("type")){
80.             case 1:
81.                 //El valor proviene de un identificador
82.                 valorDerecho = tablaDeSimbolos.valorDeUnIdentificador(
83.                         condition.getString("identifier"),
84.                         0
85.                 );
86.                 if(valorDerecho==-999999999){
87.                     agregarTextoError(
88.                             idioma.traerTexto("NoSeEncontroValorID")
89.                                     .replace("VAR",condition.getString("identifier"))
90.                     );
91.                     try {
92.                         throw new Exception();
93.                     } catch (Exception e) {
94.                         throw new RuntimeException(e);
95.                     }
96.                 }
97.                 tipoOperacion = condition.getString("operator");
98.                 break;
99.             case 2:
100.                    //El valor proviene de un número
101.                    valorDerecho = condition.getInt("value");
102.                    tipoOperacion = condition.getString("operator");
103.                    break;
104.                case 3:
105.                    //El valor proviene de un número aleatorio
106.                    valorDerecho = generarAleatorio(
107.                            condition.getJSONObject("valueFrom").getInt("from"),
108.                            condition.getJSONObject("valueFrom").getInt("to")
109.                    );
110.                    tipoOperacion = condition.getString("operator");
111.                    break;
112.                case 9:
113.                    //Pregunta si hay muro en frente
114.                    valorDerecho = resolverQueHayFrente();
115.                    break;
116.            }
117.
118.            if(Objects.equals(tipoOperacion, "<")){
```

```
119.                 return (valorIzquierdo < valorDerecho);
120.             }else if(Objects.equals(tipoOperacion, "<=")){
121.                 return (valorIzquierdo <= valorDerecho);
122.             }else if(Objects.equals(tipoOperacion, ">")){
123.                 return (valorIzquierdo > valorDerecho);
124.             }else if(Objects.equals(tipoOperacion, ">=")){
125.                 return (valorIzquierdo >= valorDerecho);
126.             }else if(Objects.equals(tipoOperacion, "==")){
127.                 return (valorIzquierdo == valorDerecho);
128.             }else if(Objects.equals(tipoOperacion, "!=")){
129.                 return (valorIzquierdo != valorDerecho);
130.             }
131.             return false;
132.         }
```

Al final de este algoritmo, se evalúan las comparaciones según correspondan.

1.2.22 Funcionalidad: Resolviendo si existe una bomba, un objeto o muro delante del personaje

Como se observó en el caso anterior, a veces se tendrá que ocupar la funcionalidad de saber qué objeto en específico tiene el personaje de frente.

Los algoritmos son sencillos y ocupan un método anteriormente descrito. Únicamente regresan falso o verdadero si coincide el número en la evaluación.

Es decir, si el método *resolverQueHayFrente*() regresa:

- El valor 1 se trata que hay un objeto fijo y el valor 4 se trata de que el personaje se encuentra en el tope máximo del mapa.
- Si el valor es 2 se trata de un objeto que el personaje puede tomar, mover o destruir libremente.
- Si el valor es 3 se trata de una bomba y es mejor ignorarlo.

El algoritmo es el siguiente:

```
1.      private boolean resolverKaboomFrente() {
2.          return resolverQueHayFrente()==3;
3.      }
4.
5.      private boolean resolverObjetoFrente() {
6.          return resolverQueHayFrente()==2;
7.      }
8.
9.      private boolean resolverMuroFrente() {
10.         return resolverQueHayFrente() == 1 || resolverQueHayFrente() == 4;
11.     }
```

1.2.23 Estructura de control: (If-Then-Else)

La primera estructura de control que se analizará será la de tipo If-Then-Else. La estructura de este algoritmo es la siguiente:

Sobre la evaluación de condiciones:

- Primero se evalúa la primera condición del primer “if”, si se cumple, se ignora el resto de la estructura (en caso de existir).

- Si la condición es falsa y continúa la estructura el siguiente paso es continuar evaluando las condiciones de un “if else” (si existen) hasta que se cumpla alguna condición. Por diseño del compilador, estas instrucciones en el objeto JSON estarán ordenadas al revés, por lo que se debe leer del último elemento al primero.
- Finalmente, si ninguna de las instrucciones se ha cumplido existe la posibilidad de que el usuario haya dejado un “else” sin condición a evaluar por lo que se deberá ejecutar al final.

Si se cumple una instrucción:

- Primero se creará un nuevo identificador a las funciones, porque cualquier variable que se pueda crear durante este subprograma solo debe coexistir dentro de él. Es decir, una variable que se declara entre un bloque if, solo deberá ser accesible dentro de ese bloque. De igual manera las variables ya declaradas las puede utilizar sin problema. Aquí se le brinda un poco de realismo al lenguaje y comienza a tener sentido la llave “*function*” de los elementos en la tabla de símbolos.
- Después se tiene que llamar a la función con la que se comienza a analizar el programa principal con las instrucciones que contiene el bloque de código de la estructura de control.
- Al final de ejecutar el “subprograma” se tienen que eliminar de la tabla de símbolos aquellas variables que pudieron ser creadas en ese bloque de código. Simplemente borrando todas aquellas que tienen el ID creado en el primer punto.
- Este método retorna el resultado de aquel análisis del subprograma. Si llegase a ser falso el resultado, se tiene que detener el análisis del programa por completo, debido a que se entiende que ha ocurrido un error y no se puede continuar.

El algoritmo del análisis del subprograma es:

```
1.      private boolean SresolverSubPrograma(JSONArray subprogram) {
2.          String IDSolucion = "subprogram" + Math.random();
3.          boolean resultado = resolverPrograma(IDSolucion,subprogram);
4.          tablaDeSimbolos.eliminarElementosPorID(IDSolucion);
5.          return resultado;
6.      }
```

El algoritmo del análisis de esta estructura de control es el siguiente:

```
1.      private boolean SI(JSONObject temporal) {
2.          boolean condicion = this.evaluarCondicion(temporal.getJSONObject("condition"));
3.          String texto =
4.                  idioma.traerTexto("InstruccionSI")+
5.                          idioma.traerTexto("Ejecutada")
6.          ;
7.          if(condicion){
8.              boolean resultado = SresolverSubPrograma(temporal.getJSONArray("subprogram"));
9.              if(!resultado){
10.                 return false;
11.             }
12.             agregarTexto(texto);
13.             return true;
```

```
14.         }else if(temporal.getJSONArray("continue").length() != 0){
15.             //Aún hay más por continuar.
16.             JSONObject tmp;
17.             for(int i = temporal.getJSONArray("continue").length() -1 ; i >= 0 ; i--){
18.                 tmp = (JSONObject) temporal.getJSONArray("continue").get(i);
19.                 //Verificando que no sea un else simple
20.                 if(Objects.equals(tmp.getString("instruction"), "SINO")){
21.                     boolean resultado = SresolverSubPrograma(tmp.getJSONArray("subprogram"));
22.                     if(!resultado){
23.                         return false;
24.                     }
25.                     agregarTexto(texto);
26.                     return true;
27.                 }
28.                 //Realiza los else if
29.                 condicion = this.evaluarCondicion(tmp.getJSONObject("condition"));
30.                 if(condicion){
31.                     boolean                                 resultado                                 =
    SresolverSubPrograma(temporal.getJSONArray("subprogram"));
32.                     if(!resultado){
33.                         return false;
34.                     }
35.                     agregarTexto(texto);
36.                     return true;
37.                 }
38.             }
39.         }
40.         return true;
41.     }
42.
```

1.2.24 Estructura de control: (Switch-Case)

La siguiente estructura de control es la que se denomina como Switch-Case, esta tiene la particularidad de evaluar un valor con casos individuales hasta encontrar con una condición válida. Pero se debe detener el análisis cuando se ejecuta un bloque (si se desea), de lo contrario seguirá evaluando hasta terminar el bloque de código. Igualmente, existe un caso por default en caso de que no se cumpla ninguna condición (o se le olvide al programador la instrucción de detener el análisis cuando se cumpla una condición) se ejecutará sin preguntar como el caso “else” simple de la estructura anterior.

En este algoritmo incluyo unas cosas de más que un lenguaje de alto nivel no se incluye (como: C, Java, entre otros), las cuales son:

- El valor que se compara debe ser una variable o ejecutar la función “*delante_de_mi*()”.
- Cuando se evalúan los elementos “case” se puede incluir antes un símbolo de comparación. Por ejemplo: “case: <10:”.
- Es obligatorio un caso por default, y al igual que en los lenguajes lenguaje de alto nivel solo se define 1 vez.

Lo anterior, para dotarle al lenguaje una mayor dinámica al lenguaje. Dependerá del educador informar sobre estas diferencias.

Algoritmo para evaluar las condiciones de la estructura de control:

```
1.      private boolean compararCondicion(int valorComparacion, JSONObject condition) {
2.          int valorDerecho = 0;
3.          String operador = condition.getString("operator");
4.          if(condition.getInt("type")==0){
5.              valorDerecho = condition.getInt("value");
6.          }else if(condition.getInt("type")==1){
7.              JSONObject tmp = condition.getJSONObject("valueFrom");
8.              valorDerecho = generarAleatorio(tmp.getInt("from"), tmp.getInt("to"));
9.          }
10.
11.         return condicionComparacionCumple(valorComparacion,operador,valorDerecho);
12.     }
13.
14.     private  boolean  condicionComparacionCumple(int  valorComparacion,  String  operador,  int
    valorDerecho) {
15.         switch (operador){
16.             case "<":
17.                 return valorComparacion < valorDerecho;
18.             case "<=":
19.                 return valorComparacion <= valorDerecho;
20.             case ">":
21.                 return valorComparacion > valorDerecho;
22.             case ">=":
23.                 return valorComparacion >= valorDerecho;
24.             case "!=":
25.                 return valorComparacion != valorDerecho;
26.             default:
27.                 return valorComparacion == valorDerecho;
28.         }
29.     }
30.
```

El algoritmo para evaluar la estructura de control:

```
1.      private boolean COMPARAR(JSONObject temporal) {
2.          int valorComparacion = 0;
3.          String texto =
4.              idioma.traerTexto("InstruccionCOMPARAR")+
5.                      idioma.traerTexto("Ejecutada")
6.          ;
7.          if(temporal.getInt("type")==1){
8.              if(tablaDeSimbolos.existeUnElemento(
9.                      temporal.getString("identifier"),
10.                     0
11.             )){
12.                 valorComparacion = tablaDeSimbolos.valorDeUnIdentificador(
13.                         temporal.getString("identifier"),
14.                         0
15.                 );
16.                 if(valorComparacion==-999999999){
17.                     agregarTextoError(
18.                             idioma.traerTexto("NoSeEncontroValorID")
19.                                     .replace("VAR",temporal.getString("identifier"))
20.                     );
21.                     return false;
22.                 }
23.             }else{
24.                 agregarTexto(texto);
25.                 agregarTextoError(
26.                         idioma.traerTexto("ElValorError")+
```

```
                                temporal.getString("identifier") +
                                idioma.traerTexto("NoPuedeContinuar")
                );
                return false;
            }
        }else if(temporal.getInt("type")==2){
            valorComparacion = this.resolverQueHayFrente();
        }

        //Evalúa las condiciones:
        JSONObject tmp = temporal.getJSONObject("cases");
        while(!tmp.isEmpty()){
            if(compararCondicion(valorComparacion,tmp.getJSONObject("condition"))){
                boolean resultado = SresolverSubPrograma(tmp.getJSONArray("subprogram"));
                if(!resultado){
                    return false;
                }
                if(tmp.getBoolean("end")){
                    agregarTexto(texto);
                    return true;
                }
            }
            tmp = tmp.getJSONObject("continue");
        }

        //Realiza el caso por default:
        tmp = temporal.getJSONObject("default");
        boolean resultado = SresolverSubPrograma(tmp.getJSONArray("subprogram"));
        if(!resultado){
            return false;
        }
        agregarTexto(texto);
        return true;
    }
```

Hay que recordar para este algoritmo que se resolverá de igual manera un subprograma con las características de las que ya se ha hablado.

1.2.25 Estructura de control: Do-While

Esta estructura ejecuta un bloque de instrucciones mientras se cumpla una condición. Además, esta instrucción tiene la característica de realizar al menos una vez el bloque de código.

La única restricción que tiene esta estructura es que no puede comparar valores de variables que se han creado en el bloque de código, como en un lenguaje formal. Por ejemplo, el siguiente algoritmo de un lenguaje de alto nivel no se podrá resolver:

```
1.          do{
2.              int i=0;
3.          }while(i<10);
```

El algoritmo es sencillo de realizar, igualmente se tiene que generar un subprograma y evaluar la condición que ha declarado el usuario.

```
1.      private boolean HACER(JSONObject temporal) {
2.          String texto =
3.                  idioma.traerTexto("InstruccionHACER")+
4.                          idioma.traerTexto("Ejecutada")
5.          ;
6.          do{
7.              boolean resultado = SresolverSubPrograma(temporal.getJSONArray("subprogram"));
8.              if(!resultado){
9.                  return false;
10.             }
11.         }while(evaluarCondicion(temporal.getJSONObject("condition")));
12.         agregarTexto(texto);
13.         return true;
14.     }
```

1.2.26 Estructura de control: While

Esta estructura es similar a la anterior, con la diferencia que para ejecutar su bloque de instrucciones siempre evaluará la condición. Las restricciones son las mismas que en la estructura anterior.

El algoritmo es el siguiente:

```
1.      private boolean REPITEHASTA(JSONObject temporal) {
2.          String texto =
3.                  idioma.traerTexto("InstruccionREPITEHASTA")+
4.                          idioma.traerTexto("Ejecutada")
5.                  ;
6.          while(evaluarCondicion(temporal.getJSONObject("condition"))){
7.              boolean resultado = SresolverSubPrograma(temporal.getJSONArray("subprogram"));
8.              if(!resultado){
9.                  return false;
10.             }
11.         }
12.         agregarTexto(texto);
13.         return true;
14.     }
```

1.2.27 Estructura de control: For

La última estructura de control es la que lleva como nombre “for”. Para esta estructura se tiene un espacio de declaración o inicialización de variables, un espacio para evaluar la condición para seguir continuando y un espacio para incrementar o decrementar variables. Estos elementos se contienen en un paréntesis, separados por un punto y coma.

Para resolver esta instrucción primero se declararán las variables necesarias o se modificarán las existentes. Si se declaran variables estas serán temporales que se borrarán de la tabla de símbolos al final de ejecutar la instrucción. Si se modifican variables ya declaradas por fuera de esta instrucción su valor modificado se conserva cuando termina la ejecución de la instrucción.

Después comenzará evaluando la condición que se declaró, si es verdadera la condición comenzará a ejecutar un nuevo subprograma. Cuando termine pasará a realizar los incrementos o decrementos de las variables según corresponda y volverá a realizar el ciclo hasta que la condición deje de cumplirse. El algoritmo es el siguiente:

```
1.      private boolean FOR(JSONObject temporal) {
2.          //Ejecutando declaraciones:
3.          String idVariablesTemporales = "for" + Math.random();
4.          //Modificando variables existentes y creando variables nuevas:
5.          JSONObject tmp ;
6.          String texto =
7.                  idioma.traerTexto("InstruccionFOR")+
8.                          idioma.traerTexto("Ejecutada")
9.                  ;
10.         ArrayList<JSONObject>          declarations          =          (ArrayList<JSONObject>)
    temporal.get("declarations");
11.         for (JSONObject declaration : declarations) {
12.             tmp = declaration;
13.             if (tmp.getInt("type") == 1) {
14.                 //Modificar Variable (Modifica tabla de símbolos)
15.                 //Buscando en Tabla de Símblos
16.                 this.MODIFICAVARIABLE(tmp);
17.             } else if (tmp.getInt("type") == 2) {
18.                 //Crear una variable temporal
19.                 tmp             =             tmp.getJSONObject("valueFrom").put("identifier",
    tmp.getString("identifier"));
20.                 this.DECLARAVARIABLE(tmp, idVariablesTemporales);
21.             }
22.         }
23.         //Evaluando Condición
24.         declarations = (ArrayList<JSONObject>) temporal.get("increments");
25.         while(evaluarCondicion(temporal.getJSONObject("condition"))){
26.             //Realizar instrucciones
27.             boolean resultado = SresolverSubPrograma(temporal.getJSONArray("subprogram"));
28.             if(!resultado){
29.                 return false;
30.             }
31.             //Realizar incrementos o decrementos
32.             for (JSONObject declaration : declarations) {
33.                 tmp = declaration;
34.                 if (tablaDeSimbolos.existeUnElemento(
35.                         tmp.getString("identifier"),
36.                         0
37.                 )) {
38.                     //El elemento a modificar fue declarado fuera del for
39.                     if (tmp.getInt("type") == 1) {
40.                         if (!(INCREMENTAVARIABLE(tmp))) {
41.                             return false;
42.                         }
43.                     } else if (tmp.getInt("type") == 2) {
44.                         if (!(DECREMENTAVARIABLE(tmp))) {
45.                             return false;
46.                         }
47.                     }
48.                 }
49.             }
50.         }
51.         tablaDeSimbolos.eliminarElementosPorID(idVariablesTemporales);
52.         agregarTexto(texto);
53.         return true;
54.     }
```

1.2.28 Llamada a función

La última funcionalidad a analizar es la llamada a función. Esta se podrá ejecutar incluso dentro de otra función.

Lo primero que se tiene que hacer es comprobar que la función se encuentre declarada, y después se tiene que comprobar que exista el mismo número de parámetros de entrada de la función declarada con la llamada a función.

Si pasa la comprobación anterior, se comenzará respaldando la tabla de símbolos actual en una pila que se ha declarado al principio. Después se creará una nueva tabla de símbolos y se tendrá que volver a analizar el programa del usuario para volver a declarar las funciones que se han definido en el programa principal.

Después de inicializar de nuevo la tabla de símbolos con las declaraciones de funciones del usuario, se tendrá que declarar las variables con su respectivo nombre y colocar esos valores en la tabla de símbolos.

Acto seguido, se ejecuta el subprograma como se ha realizado desde que se han estado viendo las estructuras de control.

Antes de finalizar la instrucción, es importante regresar la tabla de símbolos original.

El algoritmo es el siguiente:

```
1.      private boolean LLAMADAAFUNCION(JSONObject temporal) {
2.          JSONArray ListaDeFunciones = JSONPrograma.getJSONArray("functions");
3.          //Comprobando que los parámetros se han escrito bien
4.          //Buscando que la función se encuentre en la tabla de símbolos:
5.          if(!tablaDeSimbolos.existeUnElemento(temporal.getString("name"),1)){
6.              //No existe el elemento
7.              agregarTextoError(
8.                      idioma.traerTexto("FuncionNoEncontrada")+
9.                              temporal.getString("name")
10.             );
11.             return false;
12.         }
13.         //Comprobando que tenga el mismo parámetro de entrada.
14.         if(tablaDeSimbolos.valorDeUnIdentificador(temporal.getString("name"),1)==-999999999){
15.             agregarTextoError(
16.                     idioma.traerTexto("NoSeEncontroValorID")
17.                             .replace("VAR",temporal.getString("identifier"))
18.             );
19.             return false;
20.         }
21.         if(!(tablaDeSimbolos.valorDeUnIdentificador(temporal.getString("name"),1)             ==
    temporal.getJSONArray("parameter").length())){
22.             //No tienen el mismo número de parámetros
23.             agregarTextoError(
24.                     idioma.traerTexto("NumeroDeParametrosDiferente")
25.             );
26.             return false;
27.         }
28.         ///Creando nueva tabla de símbolos y guardando la anterior en una pila
29.         pilaDePilas.add(tablaDeSimbolos);
30.         tablaDeSimbolos = new TablaDeSimbolos();
31.         funcionesATablaDeSimbolos();
32.         //Declarando las variables:
```

```
33.         for(int i=0;i<ListaDeFunciones.length();i++){
34.             //Buscando la función con parámetros:
35.             if(Objects.equals(ListaDeFunciones.getJSONObject(i).getString("name"),
    temporal.getString("name"))){
36.                 JSONObject declarations = (JSONObject) ListaDeFunciones.get(i);
37.                 ArrayList<JSONObject>          list          =          (ArrayList<JSONObject>)
    declarations.get("parameter");
38.                 for(int j=0;j<temporal.getJSONArray("parameter").length();j++){
39.                     JSONObject              declaracion              =              (JSONObject)
    temporal.getJSONArray("parameter").get(j);
40.                     JSONObject tmp = list.get(j);
41.                     declaracion.put("identifier",tmp.getString("identifier"));
42.                     if(!DECLARAVARIABLE(
43.                             declaracion,
44.                             "main"
45.                     )){
46.                         return false;
47.                     }
48.                 }
49.                 //Resolviendo el programa
50.                 boolean                                    resultado                                =
    SresolverSubPrograma(declarations.getJSONArray("subprogram"));
51.                 if(!resultado){
52.                     return false;
53.                 }
54.             }
55.         }
56.         tablaDeSimbolos = pilaDePilas.pop();
57.
58.         agregarTexto(
59.                 idioma.traerTexto("InstruccionFuncion")+
60.                         temporal.getString("name")+
61.                         "() "+
62.                         idioma.traerTexto("Ejecutada")
63.         );
64.
65.         return true;
66.     }
```

Con la instrucción final se ha terminado el desarrollo de la máquina virtual.

1.3 Ejecutar la máquina virtual

Ahora se requiere de un hilo que sea capaz de ejecutar la máquina virtual. Debido a que se utilizan en varias ocasiones la instrucción "*Thread.sleep*" es necesario un hilo principal que contenga estas pausas durante la ejecución de la máquina virtual.

Comenzando a definir los objetos globales de esta clase:

Se tienen estos objetos globales de los cuales ya se habló en su momento en la definición de la clase "RunCode". Al igual que la definición anterior tendrá sus setters para que se utilicen cuando se requiera.

```
1.      private JLabel lblResultados;
2.      private JTextArea txtResultados;
3.      private int Filas;
4.      private int Columnas;
5.      private int[][] MapaBackend;
6.      private JSONObject JSONPrograma;
```

Después se tienen los siguientes objetos:

```
1.      private final RunCode runCode = new RunCode();
2.      private final cargarTexto idioma = new cargarTexto("EjecutarPrograma");
3.      private Boolean canStart;
4.      private Thread hilo;
```

Dónde:

- runCode: es el objeto que es la máquina virtual.
- idioma: Contiene todas las cadenas y configuraciones que utilizará el programa.
- canStart: es un booleano de control. Si es verdadero quiere decir que la máquina virtual se está ejecutando. Este booleano tendrá un getter para informar en a clase "Fenix" que se está ejecutando la máquina virtual en caso de ser necesario.
- hilo: es el objeto que se genera al ejecutar la máquina virtual.

1.3.1 Método "isReady"

Este método tiene como objetivo informar antes de ejecutar la máquina virtual que tiene los datos necesarios para poder ejecutar el programa del usuario, así como que no existe otra máquina virtual ejecutándose en ese momento.

```
1.      public boolean isReady() {
2.          return MapaBackend != null && JSONPrograma.getBoolean("status") && canStart;
3.      }
4.
```

1.3.2 Limpiar el mapa sin ejecutar el programa del usuario

En algunas ocasiones se necesitará "limpiar el mapa" que se muestra al usuario, usualmente siempre que se ha terminado de ejecutar el código del usuario por primera vez. Es decir, se regresa al estado inicial de cuándo el mapa fue cargado.

Solo se puede utilizar este método mientras no se esté ejecutando la máquina virtual, de lo contrario mostrará un mensaje de error.

```
1.      public void limpiarMapa() {
2.          if(canStart){
3.              runCode.cleanMap();
4.          }else{
5.              JOptionPane.showMessageDialog(
6.                      new JFrame(),
7.                      this.idioma.traerTexto("NoSePuedeLimpiar"),
8.                      this.idioma.traerTexto("LimpiarMapa"),
9.                      JOptionPane.WARNING_MESSAGE
10.             );
11.         }
12.     }
```

1.3.3 Ejecutar la máquina virtual

Este método ejecutará la máquina virtual y comenzará a ejecutar el código del usuario.

Primero se inicializa un objeto nuevo para el objeto hilo. Siempre se debe inicializar un nuevo objeto.

Si una máquina virtual se está ejecutando mostrará un mensaje de error indicando que no se puede iniciar si otra máquina virtual se está ejecutando.

```
1.      public void ejecutaPrograma(){
2.          if(canStart){
3.              hilo = new Thread(this, "EjecutarPrograma");
4.              hilo.start();
5.          }else{
6.              JOptionPane.showMessageDialog(
7.                      new JFrame(),
8.                      this.idioma.traerTexto("NoSePuedeIniciar"),
9.                      this.idioma.traerTexto("IniciarPrograma"),
10.                     JOptionPane.WARNING_MESSAGE
11.             );
12.         }
13.     }
```

1.3.4 El método Run

El método anterior lleva de la mano este método, este algoritmo inicializará de nueva forma todos los objetos necesarios en la máquina virtual. Finalmente, se ejecuta el método "*ejecutaPrograma*()" de la máquina virtual para comenzar con el análisis del programa del usuario.

Si ocurre una excepción se ignora, y al finalizar, se cambia el estado de control "*canStart*" a verdadero.

```
1.      @Override
2.      public void run() {
3.          canStart=false;
4.          runCode.setJSONPrograma(this.JSONPrograma);
5.          runCode.setTxtResultados(this.txtResultados);
6.          runCode.setLblResultados(this.lblResultados);
7.          runCode.setFilas(this.Filas);
8.          runCode.setColumnas(this.Columnas);
9.          runCode.setMapaBackend(this.MapaBackend);
10.
11.         try{
12.             runCode.ejecutaPrograma();
13.         }catch(Exception ignored){}
14.
15.         canStart=true;
16.     }
17.
```

1.3.5 Interrumpir el proceso de análisis

En ocasiones se tendrá que interrumpir el análisis del programa del usuario, sobre todo en situaciones como un "loop infinito".

Este método interrumpe el análisis del programa de la máquina virtual y muestra un mensaje si se realizó con éxito.

```
1.      public void interrumpirProceso(){
2.          if(!canStart) {
3.              hilo.stop();
4.              canStart = true;
5.              JOptionPane.showMessageDialog(
6.                      new JFrame(),
7.                      this.idioma.traerTexto("ProcesoInterrumpido"),
8.                      this.idioma.traerTexto("DetenerPrograma"),
9.                      JOptionPane.INFORMATION_MESSAGE
10.             );
11.             runCode.printHTMLMap();
12.         }else{
13.             JOptionPane.showMessageDialog(
14.                     new JFrame(),
15.                     this.idioma.traerTexto("NoSePuedeDetener"),
16.                     this.idioma.traerTexto("DetenerPrograma"),
17.                     JOptionPane.ERROR_MESSAGE
18.             );
19.         }
20.     }
```

Con esto se ha terminado de diseñar el algoritmo para ejecutar la máquina virtual.