Capítulo 7. Selección de Instrucciones

1 Objetivo de esta Fase

El objetivo de esta última fase del compilador es generar un fichero de texto con el programa en el lenguaje de salida que tenga el mismo comportamiento que el programa de entrada.

Dado el siguiente programa que se halla en el fichero *entrada.txt*:

```
DATA
	float f;
	int i;

CODE
	i = 5;
	print i / 2;   // 2

	f = 5.0;
	print (f / 2.0 + 1.0) * 2.0;  // 7.0
```

Se obtendrá el siguiente programa en *salida.txt*:

```
#source "entrada.txt"
#VAR f:float
#VAR i:int
#line 6
pusha 4
push 5
storei
#line 7
pusha 4
loadi
push 2
divi
outi
#line 9
pusha 0
pushf 5.0
storef
#line 10
pusha 0
loadf
pushf 2.0
divf
pushf 1.0
addf
pushf 2.0
mulf
outf
halt
```

Importante. Dado que en esta fase se va a concretar el código a generar es requisito previo conocer la arquitectura de destino. En el caso de este tutorial el lenguaje de salida es MAPL. Para entender este capítulo se debe leer el manual de MAPL y seguir, al menos, la primera carpeta de su tutorial *"1 Juego de Instrucciones"*.

2 Solución

2.1 Diseño. Creación de la Especificación

2.1.1 Extracción de Requisitos

En *Introducción.pdf* no están expresamente indicados los requisitos de esta fase ya que, por lo conocido de éstos, se sobreentienden. De todas formas, por completitud, se incluyen a continuación:

- La sentencia *print* deberá obtener el valor de la expresión y a continuación enviarlo a la salida estándar.
- La sentencia asignación deberá obtener el valor de la expresión de la derecha y a continuación guardarlo en la dirección de memoria que represente la expresión de la izquierda.
- El valor de una variable es el valor que contiene la dirección de memoria que representa.
- El valor de una expresión aritmética es el resultado de aplicar el operador a los valores de sus dos expresiones.

2.1.2 Metalenguaje Elegido

Dado que el objetivo de esta fase es generar código, se necesita un metalenguaje que exprese de manera precisa qué código se va a generar ante cualquier estructura del lenguaje. El metalenguaje que se usará en esta fase son las *Especificaciones de Código*.

Una especificación de código asocia *funciones de código* a las categorías sintácticas de la gramática abstracta de tal manera que al aplicarlas devuelven el código para dicha categoría. Dado que a una categoría pueden pertenecer distintos nodos, las funciones de código se definen mediante una *plantilla de código* por cada nodo de la misma[1].

2.1.3 Especificación en el Metalenguaje

En vez de comenzar la *Especificación de Código* en un documento en blanco, una alternativa más rápida es utilizar el esqueleto '*metalenguajes/Code Specification.html*' generado por *VGen*. En dicho documento aparece en la primera columna una función f_i para cada categoría y en la segunda la cabecera de una plantilla de código para cada nodo de la misma.

Una vez abierto dicho documento en Word se realizan las siguientes tareas:

- En la primera columna se cambia el nombre de las funciones (f_1, f_2, …) para que expresen mejor la labor que realiza el código que generan (run, ejecuta, valor, etc).
- En la segunda columna se completan las plantilla. Cada plantilla indica qué código genera y qué otras funciones de código invoca.

[1] La relación entre una *función de código* y sus *plantillas de código* (una por cada nodo de la categoría) recuerda a la relación entre un método de un interface Java y la implementación del mismo en cada uno de las clases que lo implementan.

El resultado es la siguiente especificación de código que expresa de manera precisa qué código se va a generar ante cada símbolo del árbol:

Función de Código	Plantillas de Código
run[[**programa**]]	run[[**programa** → *definiciones*:defVariable* *sentencias*:sentencia*]] = #SOURCE {ficheroEntrada} metadatos[[definiciones$_i$]] ejecuta[[sentencias$_i$]] HALT
metadatos[[**defVariable**]]	metadatos[[**defVariable** → *tipo*:tipo *nombre*:String]] = #VAR {nombre}: {tipoMAPL(tipo)}
ejecuta[[**sentencia**]]	ejecuta[[**print** → *expresion*:expresion]] = #LINE {end.line} valor[[expresion]] OUT$_{\text{<expresion.tipo>}}$ ejecuta[[**asigna** → *left*:expresion *right*:expresion]] = #LINE {end.line} direccion[[left]] valor[[right]] STORE$_{\text{<left.tipo>}}$
valor[[**expresion**]]	valor[[**exprAritmetica** → *left*:expresion *operador*:String *right*:expresion]] = valor[[left]] valor[[right]] si operador == "+" ADD$_{\text{<tipo>}}$ si operador == "-" SUB$_{\text{<tipo>}}$ si operador == "*" MUL$_{\text{<tipo>}}$ si operador == "/" DIV$_{\text{<tipo>}}$ valor[[**variable** → *nombre*:String]] = direccion[[variable]] LOAD$_{\text{<tipo>}}$ valor[[**literalInt** → *valor*:String]] = PUSH {valor} valor[[**literalReal** → *valor*:String]] = PUSHF {valor}
dirección[[**expresion**]]	dirección[[**variable** → *nombre*:String]] = PUSHA {definicion.direccion}

La notación *Instruccion*$_{\text{<expresión de tipo>}}$ representa a la versión adecuada de la instrucción para el tipo indicado. Ejemplo:

LOAD$_{\text{<int>}}$ → LOADI
LOAD$_{\text{<real>}}$ → LOADF

Todas las líneas con *metadatos* (prefijadas con el símbolo #) son opcionales (en los siguientes apartados se explicará su utilidad).

A continuación se definen las funciones auxiliares utilizadas en la especificación de código.

Función Auxiliar	Definición
tipoMAPL(tipo)	Si tipo == intType tipoMAPL = "int" Si tipo == realType tipoMAPL = "real"

2.1.3.1 Directivas #source, #line y #var

En la especificación de código anterior se han incluido las directivas *#source* y *#line*. Estas directivas indican de qué línea del fichero fuente proviene cada instrucción del lenguaje de salida. El incluir estas directivas es opcional, pero altamente recomendado para facilitar la depuración del código generado usando el depurador de *MAPL* (*GVM.exe*).

Para entender mejor su función basta ver lo que se obtiene si se intenta validar con *GVM* el código de un programa *que no incluye* estas directivas:

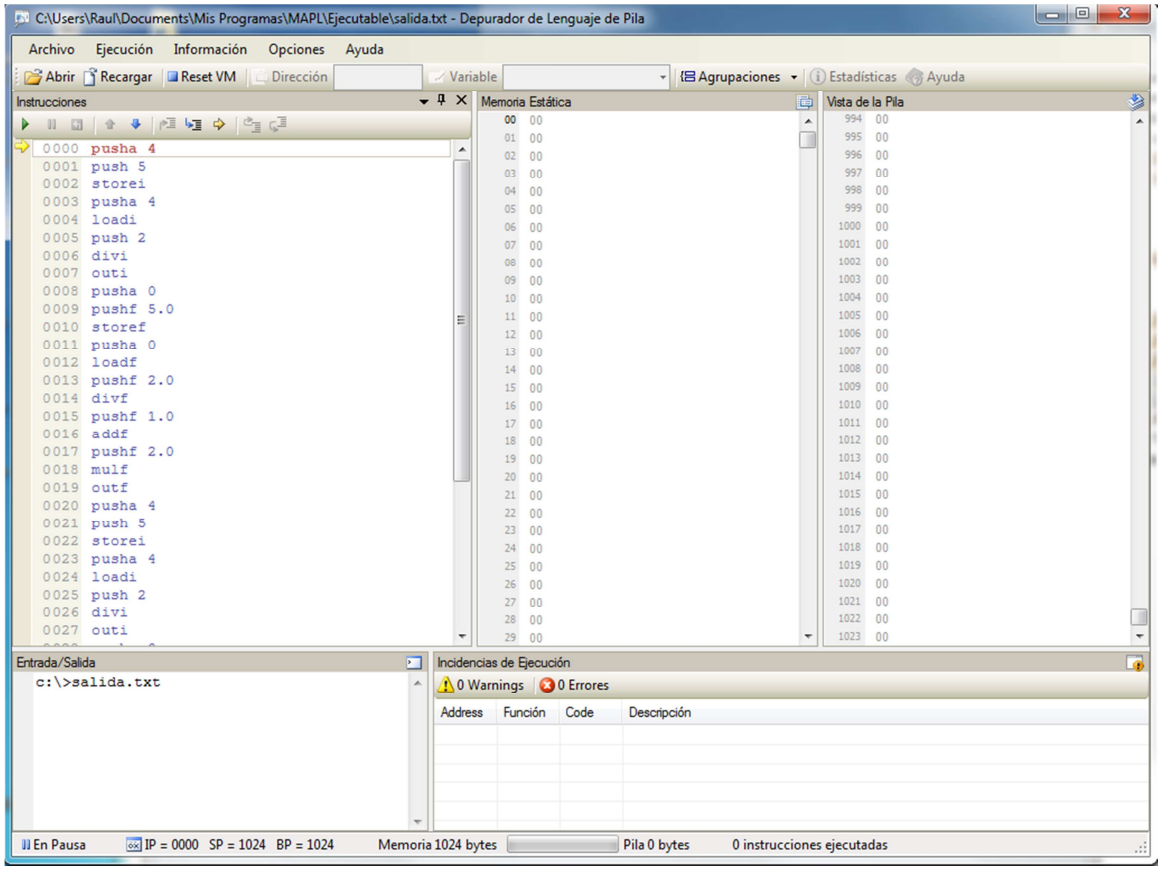

Ilustración 1. Código sin directivas

Y el mismo programa incluyendo las directivas *#source y #line*:

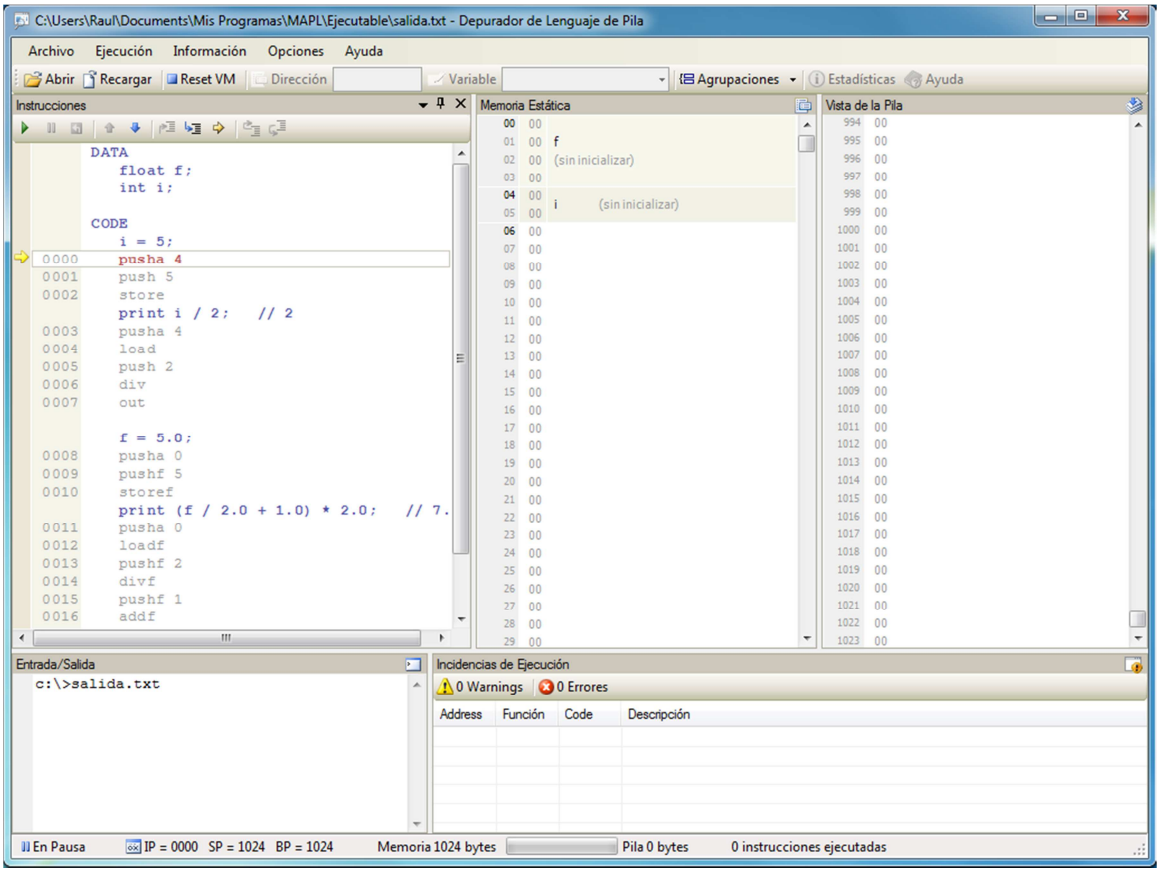

Ilustración 2. Código con *#source*, *#line* y *#var*

Al encontrar un error en el código generado es muy útil saber a qué plantilla le corresponde dicho error. Para más información sobre estas directivas se puede consultar el capítulo *"3 Uso básico del depurador"* del tutorial de *MAPL*.

2.1.3.2 Directiva #var

La directiva *#var*, al igual que ocurría con *#line* y *#source*, es opcional pero altamente recomendable. Esta directiva declara a GVM las variables del programa y sus tipos. Con esta información GVM puede:

- Dibujar las variables mostrando su posición en memoria. Pueden verse la ubicación de las variables *f* e *i* en el panel central de la ilustración 2 (las cuales no aparecían en la ilustración 1).
- Comprobar que todo acceso a memoria sea al inicio de una de las variables y que sea con una instrucción del tipo de la variable. Se detectan así escrituras a direcciones erróneas y/o con valores erróneos.

Todo lo anterior supone una gran ayuda a la depuración de los programas. Para más información sobre estas directivas se debe consultar también el capítulo *"3 Uso básico del depurador"* del tutorial de *MAPL*.

2.2 Implementación

2.2.1 Información de los Tipos

Varias instrucciones de MAPL tienen un sufijo que indican el tipo de los valores sobre los que tienen que operar (por ejemplo ADDI y ADDF). Para simplificar la generación de este tipo de instrucciones se añadirá a los tipos un método que nos dirá cuál es su sufijo. Añadimos además otro método para que cada tipo indique su nombre en la arquitectura MAPL.

```
public interface Tipo extends AST {
	int getSize();		// Este método ya estaba en el capítulo anterior
	char getSufijo();
	String getNombreMAPL();
}
```

Cada uno de estos métodos hay que redefinirlos en los tipos.

```
public class IntType extends AbstractTipo {

	public char getSufijo() {
		return 'i';
	}

	public String getNombreMAPL() {
		return "int";
	}

	// El resto igual
}
```

```
public class RealType extends AbstractTipo {

	public char getSufijo () {
		return 'f';
	}

	public String getNombreMAPL() {
		return "float";
	}

	// El resto igual
}
```

2.2.2 Selección de Instrucciones

La fase de selección de instrucciones se implementará mediante un visitor. Para implementar este *visitor*, en el proyecto de eclipse ya existe un fichero *codigo/SeleccionDeInstrucciones.java* que sugiere donde colocar esta nueva clase. En dicho fichero se tiene un esqueleto a la que solo falta añadirle los métodos *visit* para los nodos de nuestra gramática. De nuevo se puede optar por escribirlos a mano o bien copiarlos del fichero *"_PlantillaParaVisitor.txt"* generado por *VGen* en la carpeta *visitor*.

La *especificación de código* es la guía de cómo hay que implementar este *visitor*. La *plantilla de código* de un símbolo dicta la implementación del método *visit* de dicho nodo. Básicamente, cuando en la plantilla aparezca la llamada a otra función de código, se visitará dicho nodo usando su método *accept* para que así se ejecute su plantilla. Cualquier otra cosa se generará en el fichero de salida (en este caso se usa el método *genera*).

Siguiendo el método de implementación descrito, la fase de Selección de Instrucciones quedaría implementada así:

```
enum Funcion { DIRECCION, VALOR }

public class SeleccionDeInstrucciones extends DefaultVisitor {

   public SeleccionDeInstrucciones(Writer writer, String sourceFile) {
      this.writer = new PrintWriter(writer);
      this.sourceFile = sourceFile;

      instruccion.put("+", "add");
      instruccion.put("-", "sub");
      instruccion.put("*", "mul");
      instruccion.put("/", "div");
   }

   // class Programa { List<DefVariable> definiciones; List<Sentencia> sentencias; }
   public Object visit(Programa node, Object param) {
      genera("#source \"" + sourceFile + "\"");
      visitChildren(node.getDefiniciones(), param);
      visitChildren(node.getSentencias(), param);
      genera("halt");
      return null;
   }

   // class DefVariable { Tipo tipo; String nombre; }
   public Object visit(DefVariable node, Object param) {
      genera("#VAR " + node.getNombre() + ":" + node.getTipo().getNombreMAPL());
      return null;
   }

   // class Print { Expresion expresion; }
   public Object visit(Print node, Object param) {
      genera("#line " + node.getEnd().getLine());
      node.getExpresion().accept(this, Funcion.VALOR);
      genera("out", node.getExpresion().getTipo());
      return null;
   }

   // class Asigna { Expresion left; Expresion right; }
   public Object visit(Asigna node, Object param) {
      genera("#line " + node.getEnd().getLine());
      node.getLeft().accept(this, Funcion.DIRECCION);
      node.getRight().accept(this, Funcion.VALOR);
      genera("store", node.getLeft().getTipo());

      return null;
   }

   // class ExprAritmetica { Expresion left; String operador; Expresion right; }
   public Object visit(ExprAritmetica node, Object param) {
      assert (param == Funcion.VALOR);
      node.getLeft().accept(this, Funcion.VALOR);
      node.getRight().accept(this, Funcion.VALOR);
      genera(instruccion.get(node.getOperador()), node.getTipo());
      return null;
   }

   // class Variable { String nombre; }
   public Object visit(Variable node, Object param) {
      if (((Funcion) param) == Funcion.VALOR) {
         visit(node, Funcion.DIRECCION);
```

```
            genera("load", node.getDefinicion().getTipo());
        } else { // Funcion.DIRECCION
            assert (param == Funcion.DIRECCION);
            genera("pusha " + node.getDefinicion().getDireccion());
        }
        return null;
    }

    // class LiteralInt { String valor; }
    public Object visit(LiteralInt node, Object param) {
        assert (param == Funcion.VALOR);
        genera("push " + node.getValor());
        return null;
    }

    // class LiteralReal { String valor; }
    public Object visit(LiteralReal node, Object param) {
        assert (param == Funcion.VALOR);
        genera("pushf " + node.getValor());
        return null;
    }

    // Métodos auxiliares recomendados -------------------------------------
    private void genera(String instruccion) {
        writer.println(instruccion);
    }

    private void genera(String instruccion, Tipo tipo) {
        genera(instruccion + tipo.getSufijo());
    }

    private PrintWriter writer;
    private String sourceFile;
    private Map<String, String> instruccion = new HashMap<String, String>();
}
```

A la hora de implementar una especificación de código es frecuente que una categoría sintáctica (en este caso *Expresión*) tenga asociada más de una función (en este caso *valor* y *dirección*). Por tanto los nodos de esta categoría podrán tener más de una plantilla (una por función). En este caso solo *Variable* tiene dos plantillas ya que la función *dirección* no es aplicable a las demás expresiones al no ser modificables.

En general, la cuestión es cómo implementar más de una plantilla de código para un mismo nodo cuando en el *visitor* sólo se puede poner un método *visit* por nodo. Hay dos soluciones:

- Usar una clase *visitor* por cada función de código (e implementar la plantilla en el método visit de la clase que corresponda a su función).
- Usar un solo *visitor* pero usando un parámetro en el *accept* que indique qué función se quiere aplicar al nodo (y por tanto se selecciona así la plantilla correcta a ejecutar).

En este caso, por simplicidad, se ha optado por la segunda opción[2].

[2] Debido a esto, después de copiar la plantilla del *visitor*, en este caso era más cómodo dejar el código de recorrido para poder editar el parámetro del accept que dejar la llamada a *super*.

3 Ejecución

3.1 Ejecución del compilador

Se crea en *entrada.txt* el siguiente programa para probar el compilador:

```
DATA
	float f;
	int i;

CODE
	i = 5;
	print i / 2;   // 2

	f = 5.0;
	print (f / 2.0 + 1.0) * 2.0;  // 7.0
```

Y se ejecuta mediante la clase *main.Main*. Se obtiene el siguiente fichero *salida.txt*:

```
#source "entrada.txt"
#VAR f:float
#VAR i:int
#line 6
pusha 4
push 5
storei
#line 7
pusha 4
loadi
push 2
divi
outi
#line 9
pusha 0
pushf 5.0
storef
#line 10
pusha 0
loadf
pushf 2.0
divf
pushf 1.0
addf
pushf 2.0
mulf
outf
halt
```

3.2 Ejecución del programa

Para comprobar que el programa generado realmente funciona hay que ejecutarlo. Para ello hay que abrirlo con *TextVm* o *GVM*[3] (las máquinas virtuales de MAPL) y comprobar el resultado.

[3] Si en vez de abrir *salida.txt* en su posición dentro de eclipse se decide copiar éste a la carpeta de GVM es importante no olvidar ***copiar también entrada.txt*** ya que lo utiliza *#source* para la fusión con alto nivel.

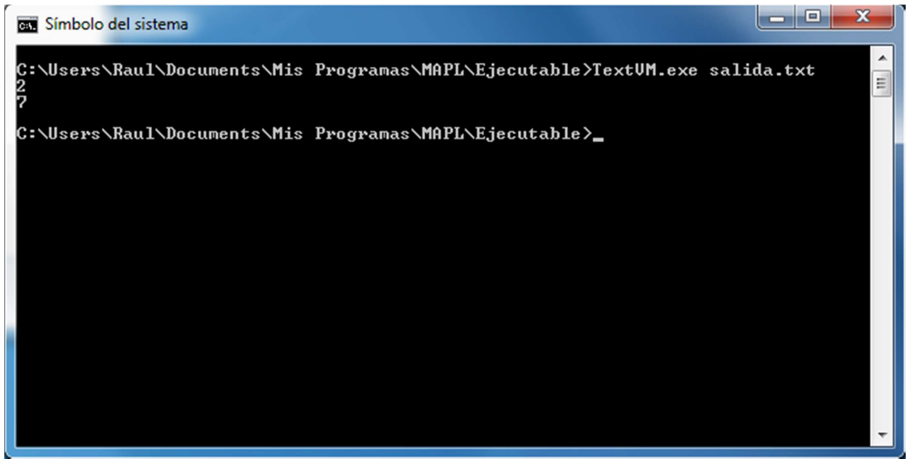

Ilustración 3. Ejecución con *TextVm*

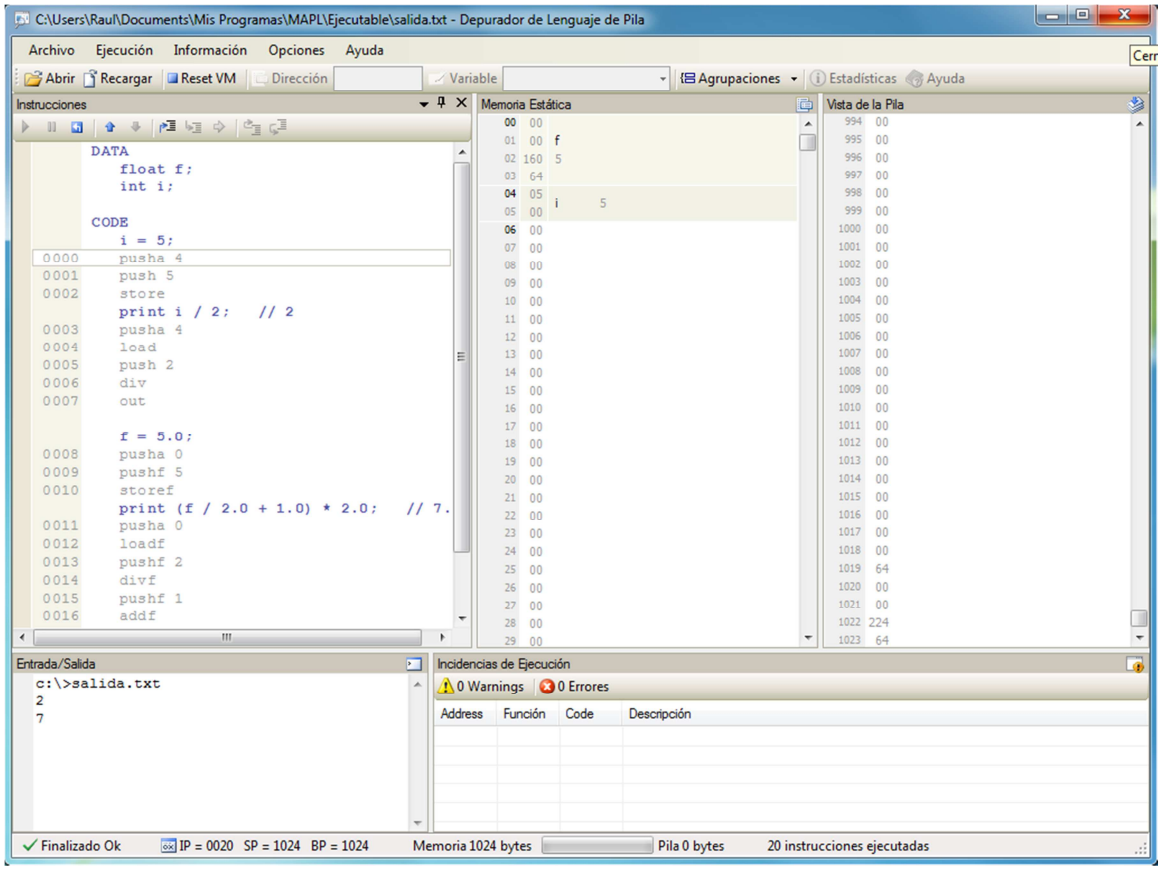

Ilustración 4. Ejecución con *GVM*

Y con esto concluiría la implementación del compilador y por tanto este tutorial.

Para cualquier sugerencia, errata encontrada o comentario puede enviarse un email a *raul@uniovi.es*

4 Resumen de Cambios

Fichero	Acción	Descripción
Code Specification.html	Creado	Metalenguaje con el que se ha descrito qué hay que generar para cada estructura del lenguaje
Tipo.java	Modificado	Se añaden las propiedades de solo lectura *sufijo* y *nombreMAPL*
IntType.java	Modificado	Implementación de *getSufijo* y *getNombreMAPL*
RealType.java	Modificado	Implementación de *getSufijo* y *getNombreMAPL*
SelecciónDeInstrucciones.java	Modificado	Visitor que implementa las funciones del código de la especificación de código
entrada.txt	Modificado	Programa para probar la generación de código
salida.txt	Creado	Código generado a partir de *entrada.txt*