

## Capítulo 7. Selección de Instrucciones

### 1 Objetivo de esta Fase

El objetivo de esta última fase del compilador es generar un fichero de texto que contenga un 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 *source.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 *output.txt*:

```
#source "source.txt"
#GLOBAL f:float
#GLOBAL 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 Trabajo autónomo del alumno

Diseñar e implementar esta fase *antes de seguir leyendo* el resto del documento. Una vez hecho, comparar la solución del alumno con la planteada a continuación.

## 3 Solución

### 3.1 Diseño. Creación de la Especificación

#### 3.1.1 Extracción de Requisitos

En *Descripción del Lenguaje.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 contenido de la dirección de memoria que representa.
- El valor de una expresión aritmética es el resultado de aplicar el operador indicado a los valores de sus dos expresiones.

#### 3.1.2 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 es una *Especificación 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<sup>1</sup>.

#### 3.1.3 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* (o editor equivalente) se realizarán las siguientes tareas:

- En la primera columna, se cambia el nombre de las funciones ( $f_1, f_2, \dots$ ) para que expresen mejor la labor que realiza el código que generan (*run, ejecuta, valor, etc.*).
- En la segunda columna, se completará las plantillas. Cada plantilla deberá indicar qué código genera y qué otras funciones de código invoca.

---

<sup>1</sup> 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[[ <b>program</b> ]]	run[[ <b>programa</b> → <i>definitions</i> :varDefinition* <i>sentences</i> :sentence* ]] = #SOURCE {sourceFile} metadata[[ <i>definitions</i> <sub>i</sub> ]] execute[[ <i>sentences</i> <sub>i</sub> ]] HALT
metadata[[ <b>varDefinition</b> ]]	metadata[[ <b>varDefinition</b> → <i>type</i> :type <i>name</i> :String ]] = #GLOBAL {name}: {MAPLType( <i>type</i> )}
execute[[ <b>sentence</b> ]]	execute[[ <b>print</b> → <i>expression</i> :expression ]] = #LINE {end.line} value[[ <i>expression</i> ]] OUT< <i>expression.type</i> > execute[[ <b>assignment</b> → <i>left</i> :expression <i>right</i> :expression ]] = #LINE {end.line} address[[ <i>left</i> ]] value[[ <i>right</i> ]] STORE< <i>left.type</i> >
value[[ <b>expression</b> ]]	value[[ <b>arithmeticExpression</b> → <i>left</i> :expression <i>operator</i> :String <i>right</i> :expression ]] = value[[ <i>left</i> ]] value[[ <i>right</i> ]] si operador == "+" ADD< <i>arithmeticExpression.type</i> > si operador == "-" SUB< <i>arithmeticExpression.type</i> > si operador == "*" MUL< <i>arithmeticExpression.type</i> > si operador == "/" DIV< <i>arithmeticExpression.type</i> > value[[ <b>variable</b> → <i>name</i> :String ]] = address[[ <i>variable</i> ]] LOAD< <i>variable.type</i> > value[[ <b>intConstant</b> → <i>value</i> :String ]] = PUSH { <i>value</i> } value[[ <b>realConstant</b> → <i>value</i> :String ]] = PUSHF { <i>value</i> }
address[[ <b>expression</b> ]]	address[[ <b>variable</b> → <i>name</i> :String ]] = PUSHA { <i>variable.definition.address</i> }

La notación *Instruccion*<*expresión de tipo*> representa a la versión adecuada de la instrucción para el tipo indicado. Ejemplos:

LOAD<int>      →      LOADI  
LOAD<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
MAPLType(type)	Si type == intType MAPLType = "int" Si type == realType MAPLType = "float"

### 3.1.3.1 Directivas #source, #line y #global

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 en la imagen siguiente lo que se obtiene si se intenta validar con *GVM* el código de un programa *que no incluye* estas directivas:

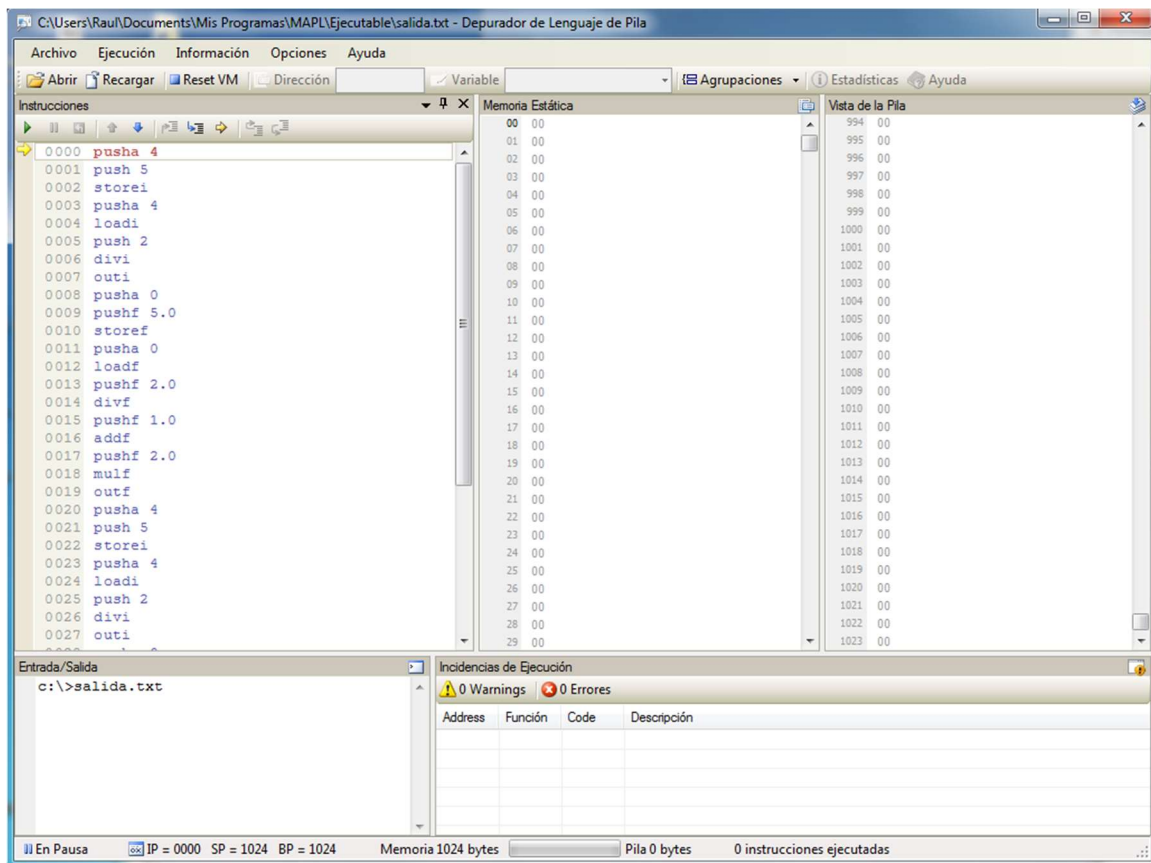


Ilustración 1. Código sin directivas

Se muestra ahora una imagen con el mismo programa incluyendo las directivas *#source* y *#line*:

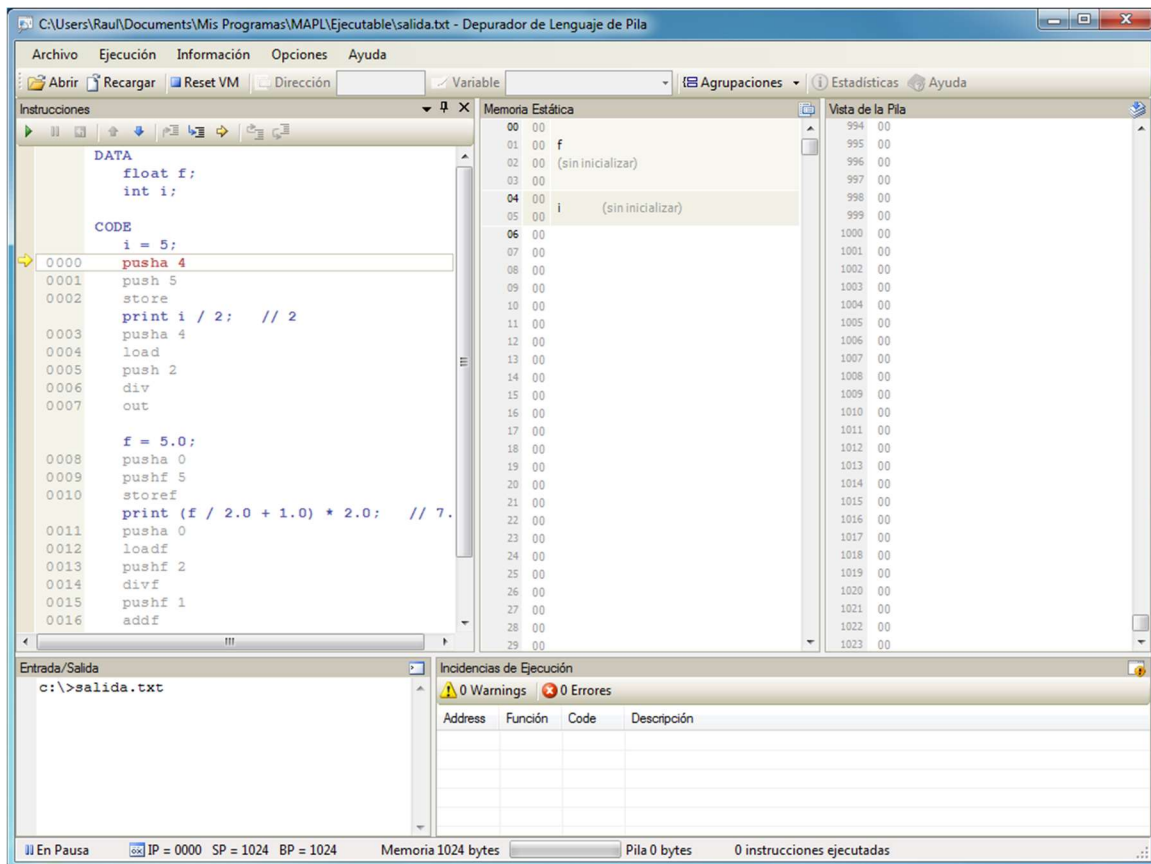


Ilustración 2. Código con *#source*, *#line* y *#global*

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 "4 Metadatos" del tutorial de MAPL.

### 3.1.3.2 Directiva *#global*

La directiva *#global*, 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 hacer dos cosas:

- *Dibujar* las variables mostrando su posición en memoria. Puede 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 podrán detectar 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.

## 3.2 Implementación

### 3.2.1 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 Type extends AST {  
    public int getSize();    // Este método ya estaba en el capítulo anterior  
  
    char getSuffix();  
  
    String getMAPLName();  
}
```

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

```
public class IntType extends AbstractType {  
  
    @Override  
    public char getSuffix() {  
        return 'i';  
    }  
  
    @Override  
    public String getMAPLName() {  
        return "int";  
    }  
  
    // El resto igual...  
}
```

```
public class RealType extends AbstractType {  
  
    @Override  
    public char getSuffix() {  
        return 'f';  
    }  
  
    @Override  
    public String getMAPLName() {  
        return "float";  
    }  
  
    // El resto igual...  
}
```

### 3.2.2 Selección de Instrucciones

La fase de selección de instrucciones se implementará mediante un visitor. Para implementarlo, en el proyecto ya existe un fichero “*codegeneration/CodeSelection.java*” que indica 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*. Cada plantilla de código dicta la implementación del método *visit* de su nodo. Básicamente, consiste en:

- Cuando en la plantilla aparezca la llamada a otra función de código, ésta se implementará visitando dicho nodo con *accept* (y ejecutar así su plantilla)<sup>2</sup>.
- Cualquier otra cosa que aparezca en la plantilla se escribirá tal cual en el fichero de salida (en este caso se usa el método auxiliar *out(String)*).

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

```
public class CodeSelection extends DefaultVisitor {

    private Map<String, String> instruccion = new HashMap<String, String>();

    public CodeSelection(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 Program { List<VarDefinition> definitions; List<Sentence> sentences; }
    public Object visit(Program node, Object param) {
        out("#source \"" + sourceFile + "\"");
        visitChildren(node.getDefinitions(), param);
        visitChildren(node.getSentences(), param);
        out("halt");
        return null;
    }

    // class VarDefinition { Type type; String name; }
    public Object visit(VarDefinition node, Object param) {
        out("#GLOBAL " + node.getName() + ":" + node.getType().getMAPLName());
        return null;
    }

    // class Print { Expression expression; }
    public Object visit(Print node, Object param) {
        line(node);
        node.getExpression().accept(this, CodeFunction.VALUE);
        out("out", node.getExpression().getType());
        return null;
    }
}
```

<sup>2</sup> En esta solución, cuando hay que recorrer todos los hijos de una lista, para abreviar se usa el método auxiliar *visitChildren* heredado de *DefaultVisitor*. Este no es más que un bucle *for* en el que se invoca el *accept* de todos los hijos.

```
// class Assignment { Expression left; Expression right; }
public Object visit(Assignment node, Object param) {
    line(node);
    node.getLeft().accept(this, CodeFunction.ADDRESS);
    node.getRight().accept(this, CodeFunction.VALUE);
    out("store", node.getLeft().getType());
    return null;
}

// class ArithmeticExpression { Expression left; String operator; Expression right;
}
public Object visit(ArithmeticExpression node, Object param) {
    assert (param == CodeFunction.VALUE);
    node.getLeft().accept(this, CodeFunction.VALUE);
    node.getRight().accept(this, CodeFunction.VALUE);
    out(instruccion.get(node.getOperator()), node.getType());
    return null;
}

// class Variable { String name; }
public Object visit(Variable node, Object param) {
    if (((CodeFunction) param) == CodeFunction.VALUE) {
        visit(node, CodeFunction.ADDRESS);
        out("load", node.getType());
    } else { // Funcion.DIRECCION
        assert (param == CodeFunction.ADDRESS);
        out("pusha " + node.getDefinicion().getAddress());
    }
    return null;
}

// class IntConstant { String valor; }
public Object visit(IntConstant node, Object param) {
    assert (param == CodeFunction.VALUE);
    out("push " + node.getValue());
    return null;
}

// class RealConstant { String valor; }
public Object visit(RealConstant node, Object param) {
    assert (param == CodeFunction.VALUE);
    out("pushf " + node.getValue());
    return null;
}

// # -----
// Métodos auxiliares recomendados (opcionales) -----

// Imprime una línea en el fichero de salida
private void out(String instruction) {
    writer.println(instruction);
}

private void out(String instruccion, Type tipo) {
    out(instruccion + tipo.getSuffix());
}

private void line(AST node) {
    line(node.getEnd());
}
```



```
private void line(Position pos) {  
    if (pos != null)  
        out("\n#line " + pos.getLine());  
    else  
        System.out.println("#Line no generado. Se ha pasado una Position null. Falta i  
nformación en el AST");  
}  
  
private PrintWriter writer;  
private String sourceFile;  
}
```

A la hora de implementar una especificación de código es frecuente que una categoría sintáctica (en este caso *Expression*) 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 cada 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.

## 4 Ejecución

### 4.1 Ejecución del compilador

Se crea en *source.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 *output.txt*:

```
#source "source.txt"  
#GLOBAL f:float  
#GLOBAL 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
```

## 4.2 Ejecución del programa

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

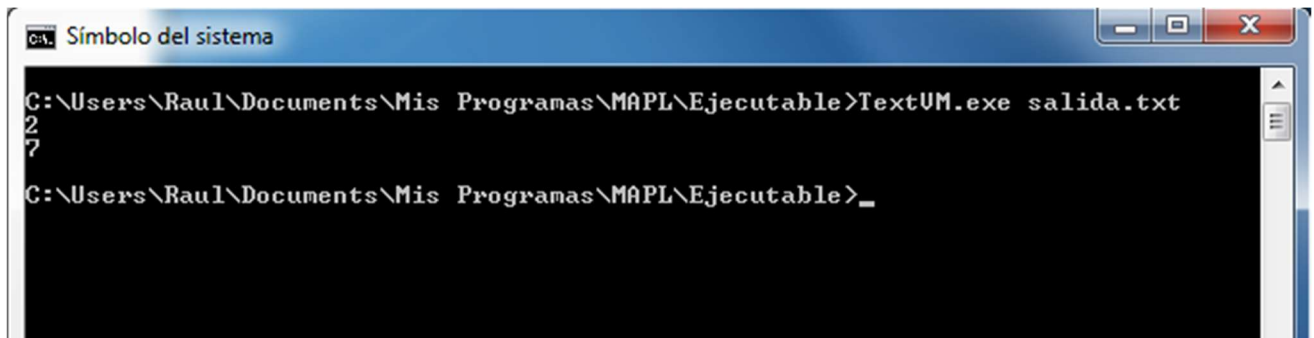


Ilustración 3. Ejecución con *TextVm*

<sup>3</sup> Si, en vez de abrir *output.txt* en la ubicación donde se ha generado, se decide copiar éste a la carpeta de GVM, es importante no olvidar **copiar también *source.txt*** ya que lo utiliza *#source* para la fusión con alto nivel.

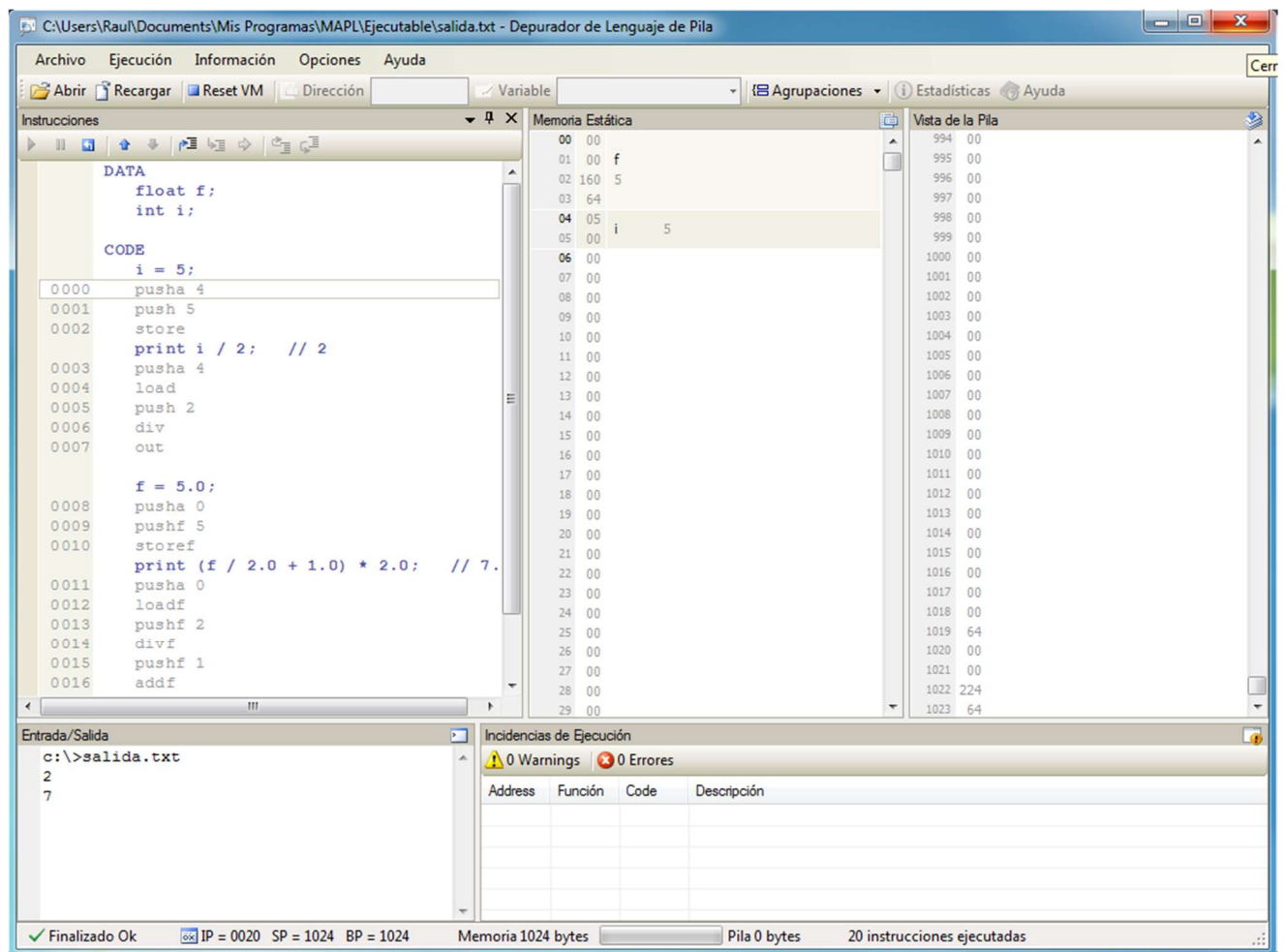


Ilustración 4. Ejecución con GVM

## 5 Resumen de Cambios

Fichero	Acción	Descripción
<b>Code Specification.pdf</b>	Creado	Metalenguaje con el que se ha descrito qué hay que generar para cada estructura del lenguaje
<b>Type.java</b>	Modificado	Se añaden las propiedades de solo lectura <i>sufijo</i> y <i>nombreMAPL</i>
<b>IntType.java</b>	Modificado	Implementación de <i>getSufijo</i> y <i>getNombreMAPL</i>
<b>RealType.java</b>	Modificado	Implementación de <i>getSufijo</i> y <i>getNombreMAPL</i>
<b>CodeSelection.java</b>	Modificado	Visitor que implementa las funciones del código de la especificación de código
<b>source.txt</b>	Modificado	Programa para probar la generación de código
<b>output.txt</b>	Creado	Código generado a partir de <i>source.txt</i>

Y con esto concluiría la implementación del compilador y, por tanto, este tutorial. Para cualquier sugerencia, errata o comentario, puede enviarse un email a [raul@uniovi.es](mailto:raul@uniovi.es).