



Máquina Abstracta MAPL (v1.3.1)

Raúl Izquierdo Castanedo
raul@uniovi.es

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática

Escuela de Ingeniería Informática de Oviedo
Universidad de Oviedo



Para todos los componentes
incluidos junto con este
documento

Máquina Abstracta MAPL

Contenido

1	Introducción	3
1.1	Descripción de MAPL.....	3
1.2	Ejecución de un programa MAPL	3
2	Funciones Principales	5
2.1	Objetivos	5
2.2	Funcionalidades para enseñar.....	5
2.2.1	Enseñar el juego de instrucciones interactivamente	5
2.2.2	Mostrar la organización de la memoria dinámicamente	6
2.3	Funcionalidades para corregir.....	8
2.3.1	Verificación mediante Comprobaciones Semánticas.....	8
2.3.2	Reproducción de Errores.....	9
2.3.3	Fusión con Alto Nivel.....	10
3	Tutorial	12
3.1	Estructura del tutorial	12
3.2	Ejecución de cada ejemplo.....	12
4	Metadatos	13
4.1	Directivas para la declaración de variables globales.....	13
4.2	Directivas para la declaración de Funciones	14
4.3	Declaración de Tipos de Usuario.....	15
4.4	Directivas para la Fusión con alto nivel.....	16
4.5	Tamaño de la memoria	16
5	Arquitectura de MAPL.....	17
6	Apéndice. Juego de Instrucciones	18

1 Introducción

1.1 Descripción de MAPL

MAPL es una Máquina Abstracta creada como herramienta para la asignatura *Diseño de Lenguajes de Programación* de la *Escuela de Ingeniería Informática* de la *Universidad de Oviedo*. Se diseñó para ser el código destino del compilador que se desarrolla como práctica obligatoria en dicha asignatura.

La máquina abstracta MAPL está implementada en dos máquinas virtuales:

- **TextVM.exe** (*Text Virtual Machine*). Es un intérprete de MAPL en línea de comando. Haciendo una analogía con Java, sería el equivalente a *java.exe*.
- **GVM.exe** (*Graphical Virtual Machine*). Es un depurador gráfico de MAPL. Permite ejecutar programas paso a paso y comprobar los cambios en el estado de la máquina (memoria estática, pila y registros).

1.2 Ejecución de un programa MAPL

En el fichero *ejemplo.txt* se encuentra un programa que se limita a pedir dos números al usuario e imprimir su suma¹:

```
' - Pulse F7 para ejecutar cada instrucción  
' - Pulse F6 para retroceder una instrucción  
  
in  
in  
add  
out
```

Para ejecutar dicho programa con *TextVM* basta con invocarlo pasándole el nombre del fichero (no hay opciones en línea de comando):

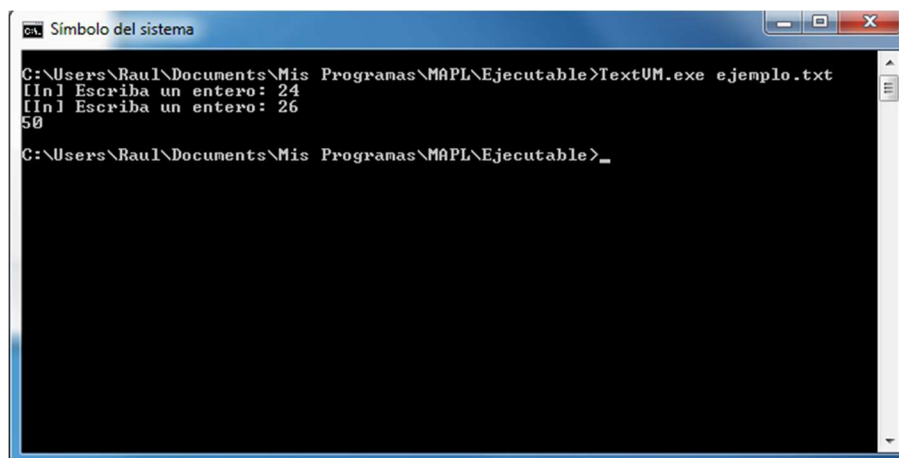


Ilustración 1. Ejecución con TextVM

¹ Las dos primeras líneas, al empezar con una comilla, son comentarios que se ignoran en la ejecución.

Para ejecutar el mismo programa con el depurador, basta con abrir GVM y seleccionar dicho fichero en la ventana inicial que aparecerá:

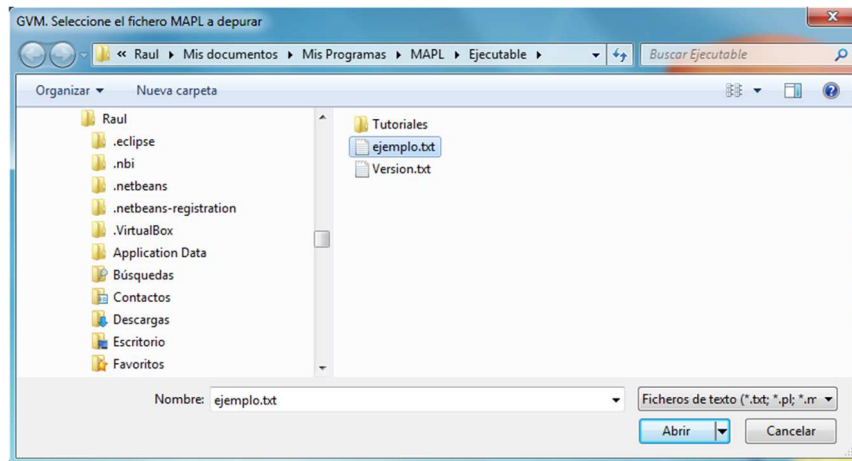


Ilustración 2. Seleccionar programa con GVM

Una vez abierto el programa se puede ejecutar de varias maneras:

- F5 para ejecutar el programa completo (como se hizo en *TextVM*).
- F7 para ejecutar una sola instrucción.
- F6 para retroceder una instrucción.

Estos son algunos de los comandos de GVM. El resto se pueden ver en los menús de la aplicación (fundamentalmente en el menú *Ejecución*).

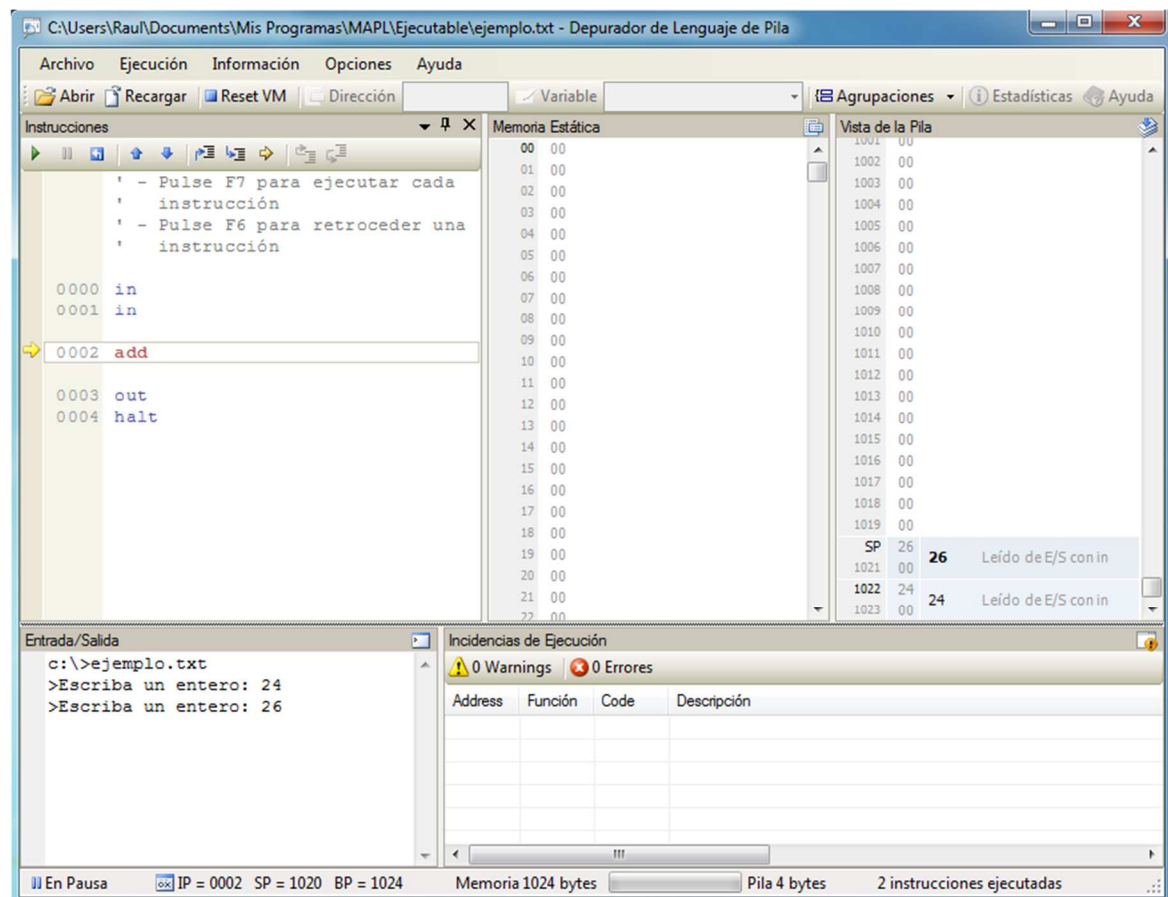


Ilustración 3. Ejecución paso a paso con GVM

2 Funciones Principales

2.1 Objetivos

MAPL se diseñó con dos objetivos fundamentales:

- **Enseñar.** Facilitar el aprendizaje de los conceptos teóricos de las fases de *gestión de memoria* y *generación de código* de la construcción de un compilador.
- **Corregir.** Una vez aprendidos los conceptos teóricos hay que ponerlos en práctica mediante la implementación de un compilador. Una vez finalizada la implementación de éste, lo que necesita el alumno es:
 1. Saber en el menor tiempo posible *si ha generado el código correctamente*.
 2. Y si no es así, necesita saber en el menor tiempo posible *qué tiene que cambiar*.

En los apartados siguientes se mostrarán algunas de las funcionalidades de MAPL². Cada una de ellas se plantea como respuesta a uno de los dos objetivos establecidos anteriormente:

1. Funcionalidades para lograr el objetivo de *enseñar*
 - Enseñar el juego de Instrucciones interactivamente
 - Mostrar la organización de la memoria dinámicamente
2. Funcionalidades para lograr el objetivo de *corregir*
 - Verificación mediante comprobaciones semánticas
 - Reproducción de los errores
 - Fusión con alto nivel

2.2 Funcionalidades para enseñar

2.2.1 Enseñar el juego de instrucciones interactivamente

GVM ofrece una manera visual de aprender el juego de instrucciones de la máquina. Ejecutando con GVM la primera carpeta del tutorial ("*1 Juego de Instrucciones*") se podrá probar lo que hace cada una de las instrucciones del lenguaje directamente sobre la máquina (ilustración 4).

² El resto podrán verse en el tutorial

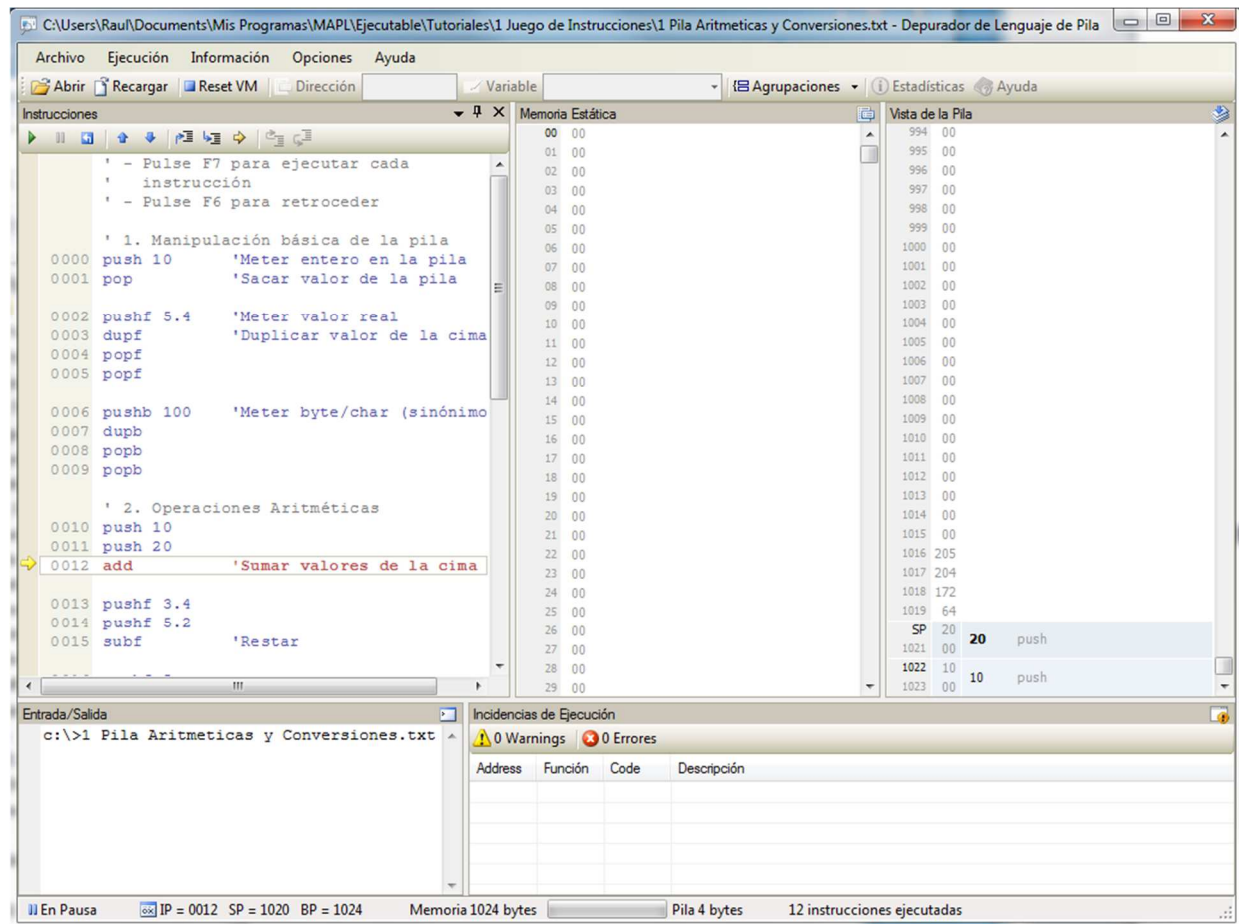


Ilustración 4. Ejercicio en el momento de mostrar la instrucción *add*

2.2.2 Mostrar la organización de la memoria dinámicamente

GVM da soporte a la enseñanza teórica de la fase de *gestión de memoria* de un compilador mostrando, entre otras cosas, dónde se ubican las variables locales y los parámetros y cómo se realiza la disposición en memoria de los tipos compuestos (vectores y estructuras). Se muestra también cómo se realiza el direccionamiento relativo de las variables (ilustraciones 5 y 6).

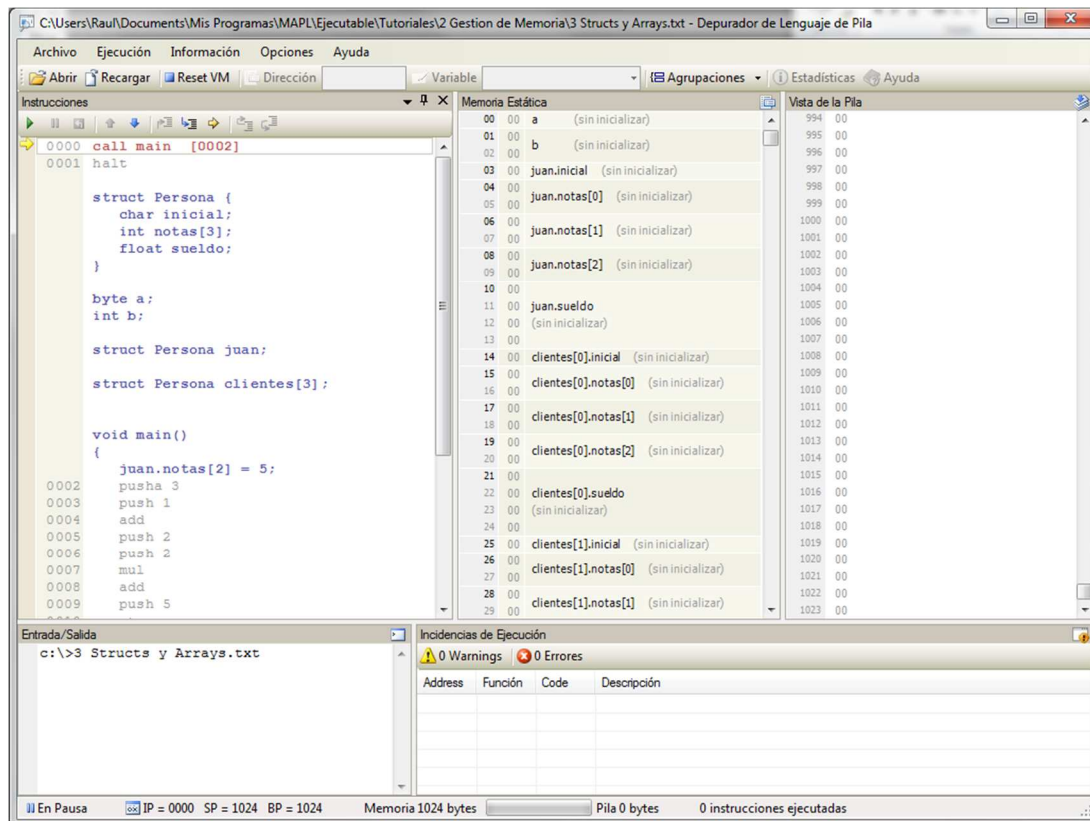


Ilustración 5. En el panel central (*Memoria Estática*) se muestra cómo se disponen las variables globales

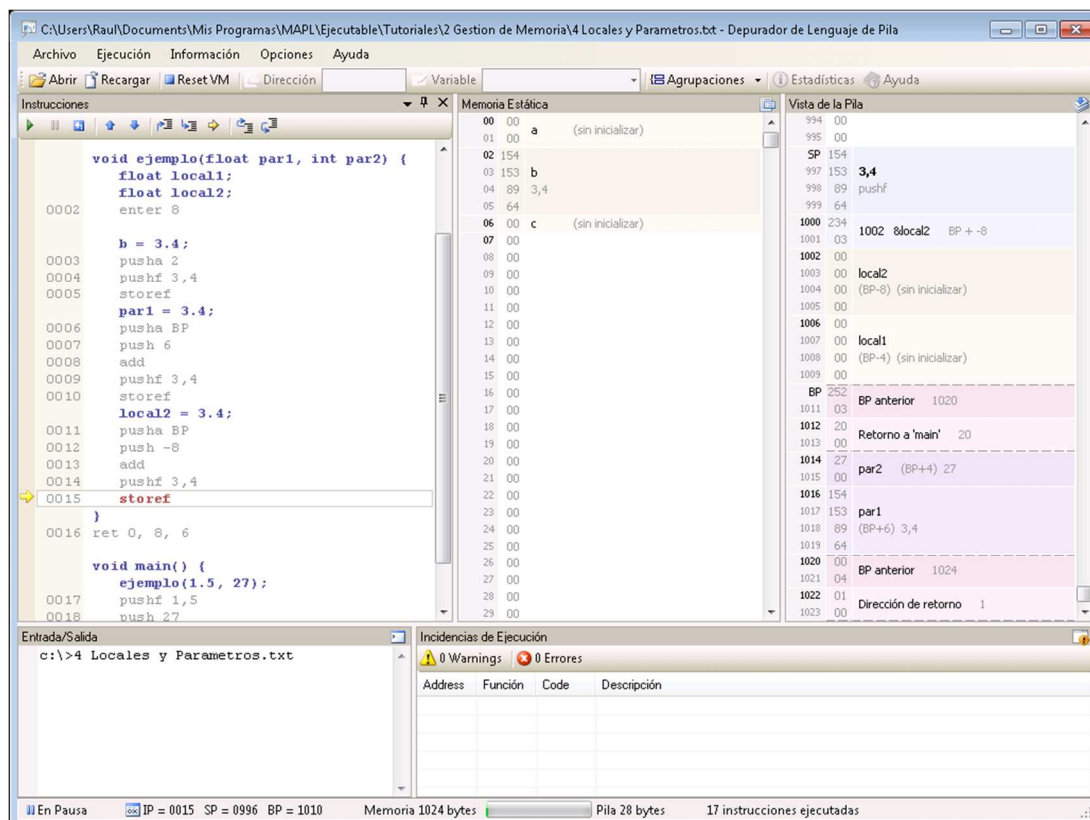


Ilustración 6. En el panel de la derecha (*Vista de Pila*) se muestran las variables locales y los parámetros junto con sus direcciones relativas (BP +/- constante)

2.3 Funcionalidades para *corregir*

A la hora de que el alumno pase a verificar el compilador que ha creado, MAPL ayudará dando soporte directo a las tres tareas que tendrá que realizar en dicho momento:

1. Detectar lo más rápidamente posible *si hay errores* en el código generado por su compilador. Para ayudar en ello, tanto *TextVM* y *GVM* incluyen verificación mediante *comprobaciones semánticas*.
2. Y si hay algún error, intentar que el alumno descubra en el mínimo tiempo posible *dónde* está el error y *por qué* es un error. Para ayudar en ello *GVM* permite la *reproducción de errores*.
3. Y una vez que entienda el por qué su código está mal, determine lo más rápidamente posible *qué parte* de su compilador tiene que modificar para generar el código correcto. Para ayudar en ello *GVM* soporta la *fusión con alto nivel*.

2.3.1 Verificación mediante Comprobaciones Semánticas

Que un programa finalice sin errores no significa que sea correcto. Por ejemplo, el siguiente programa finalizaría con normalidad y aparentemente sin errores incluso dejando la pila vacía. Sin embargo, durante su ejecución se ha corrompido la pila³.

```
call f
halt

f:
push 1
push 0
ret 0,0,4
```

El alumno no solo quiere saber *si la finalización del programa* es correcta (que en el caso anterior lo es) sino *si lo que ha generado su compilador* ha sido correcto (si ha generado las instrucciones adecuadas).

Para facilitar en esta tarea, *TextVM* y *GVM* incorporan más de un centenar de comprobaciones semánticas que detectan secuencias de código mal generado. Dado un programa generado por el compilador del alumno basta con ejecutarlo y observar si han encontrado situaciones anómalas⁴.

Una muestra de las situaciones anómalas que detectan estas comprobaciones son:

- Usar una operación sobre operandos de otro tipo o sobre valores que no son operandos (no se puede operar con el BP guardado, con memoria de las variables locales, etc.).
- Accesos a memoria que no se corresponden con el inicio de una variable o con su tipo (incluyendo accesos a vectores y estructuras).
- Funciones con caminos que no acaban en *ret* (y por tanto la ejecución continúa inesperadamente en otra parte de la misma función o de otra).

³ Para saber más detalles de qué está mal solo hay que ejecutarlo con *GVM* y éste resaltaré la línea del error y el por qué lo es.

⁴ *GVM* y *TextVM* realizan tanto comprobaciones estáticas como dinámicas. Las comprobaciones estáticas se realizan al cargar el programa mediante una ejecución abstracta previa donde se predicen los errores *antes* de que se produzcan. Las comprobaciones dinámicas se realizan durante la ejecución del programa y detectan errores que ya se han producido.

- Código muerto (no accesible). Posible síntoma de haber generado mal los saltos.
- Detección de pila corrompida. En cada instrucción se detecta si ha dejado basura en la pila y/o ha retirado bytes de más. Además, al finalizar cada función, se comprueba que la pila esté tal y como estaba al entrar.
- Detección de que los tamaños de variables locales, parámetros y tipos de retornos sean coincidentes con los valores de los *ret* y el *enter* de cada función.

2.3.2 Reproducción de Errores

Una vez que se sabe que el código generado no está correcto lo que se necesita es saber el *porqué* está mal dicho código para poder determinar cuál hubiera sido el código correcto.

La verificación semántica del apartado anterior notifica mediante un *warning* cada situación incorrecta que detecta:

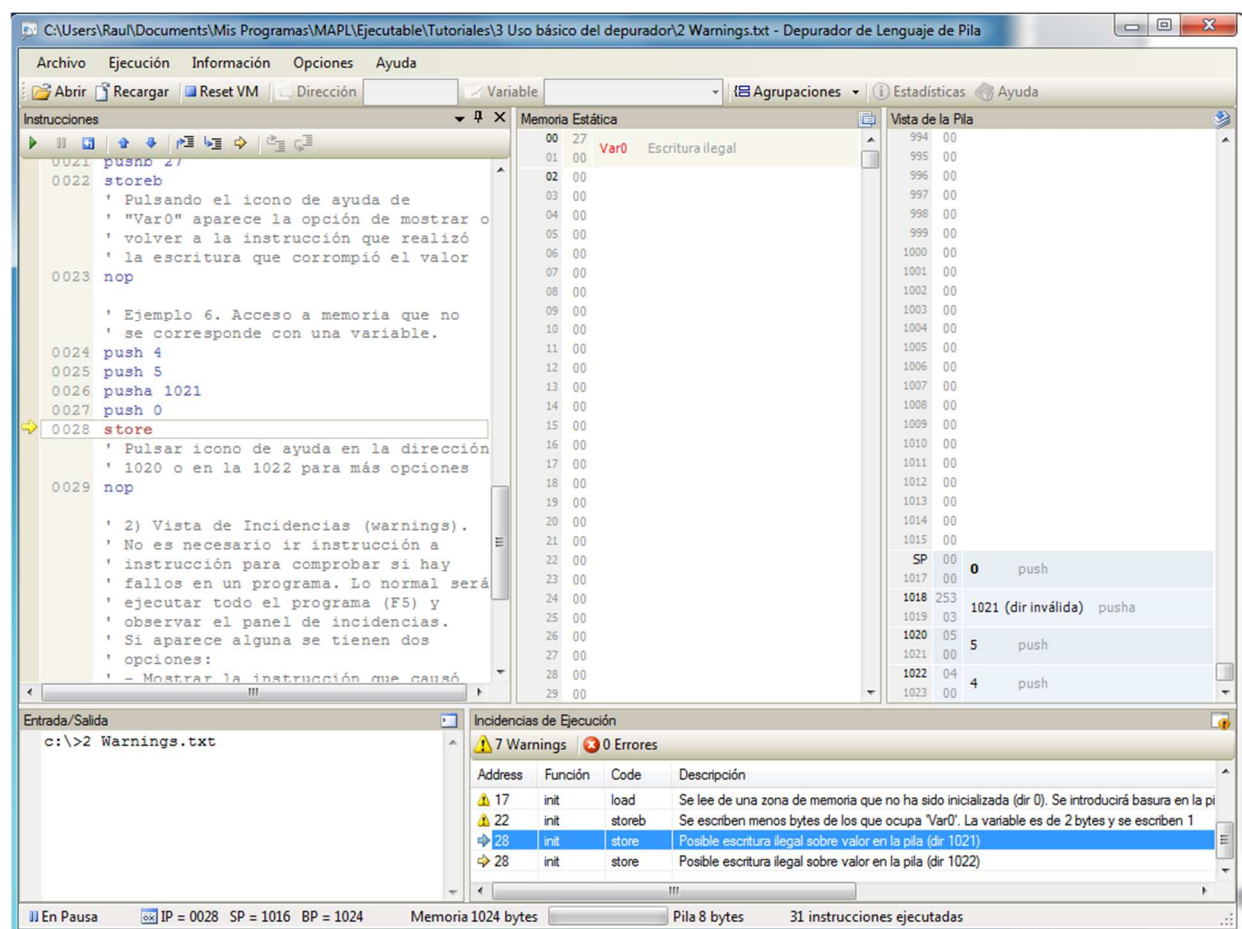


Ilustración 7. Warnings en panel inferior: rebobinando hasta el momento del tercer warning

Haciendo *doble-click* sobre un *warning*, GVM rebobina al momento previo a la ejecución de la instrucción sospechosa, mostrando el estado de la máquina tal y como estaba en ese momento (memoria estática, pila y registros). Al reproducir el error de esta manera se puede comprobar, por ejemplo, qué operandos va a tomar de la pila o en qué dirección va a escribir. Con ello se facilita el determinar si sobra o falta alguna instrucción en ese punto y por tanto averiguar por qué falla el programa.

Se puede continuar además la depuración en ese punto paso a paso tanto hacia adelante como hacia atrás (esto último es útil para averiguar, por ejemplo, cómo ha llegado a la pila un determinado valor).

El hecho de haber rebobinado hasta un *warning* y haber ejecutado instrucciones desde ese punto no impide ir directamente a la reproducción de otro error seleccionando su *warning* en la lista.

2.3.3 Fusión con Alto Nivel

Finalmente, una vez encontradas las instrucciones incorrectas, queda saber *qué parte del compilador hay que modificar* (al fin y al cabo, el programa que hay que corregir es el compilador; no el programa que éste ha generado).

Para ello *GVM* ofrece directivas que permiten asociar cada instrucción *MAPL* con el código de alto nivel a partir del cual fue generada. De esta manera es inmediato saber qué plantilla de código del compilador es la que está generando las instrucciones incorrectas.

En el ejemplo siguiente se puede ver que la instrucción *add* ha sido generada por la segunda asignación de la función *inicia*.

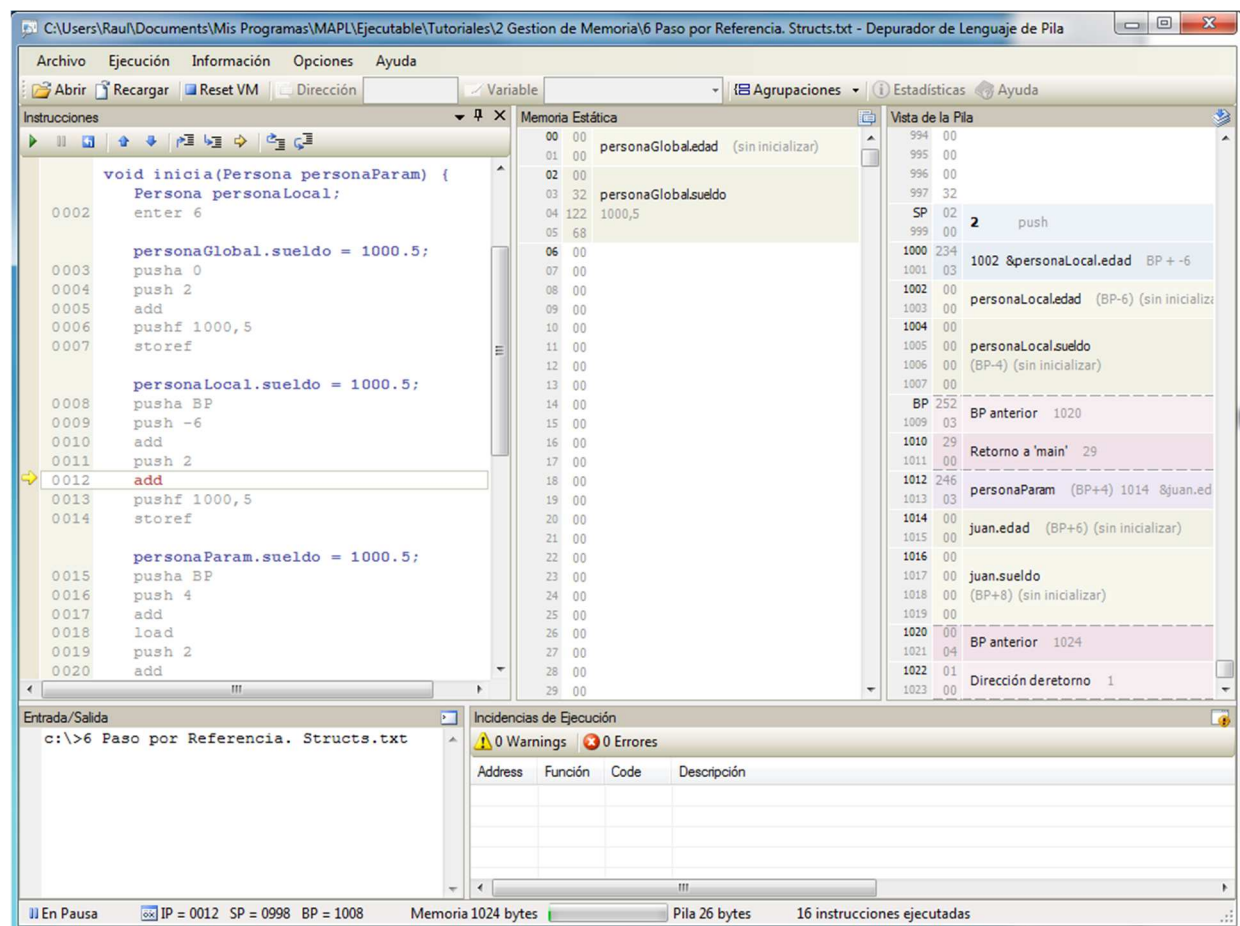


Ilustración 8. En el panel de la izquierda se muestra las instrucciones generadas por cada sentencia

Aunque la fusión con alto nivel es opcional, es altamente recomendable. Véase el mismo programa sin fusión de alto nivel. Es más difícil saber a qué parte del compilador es la que ha generado la misma instrucción *add* debido a que el código de todas las sentencias está sin delimitar.

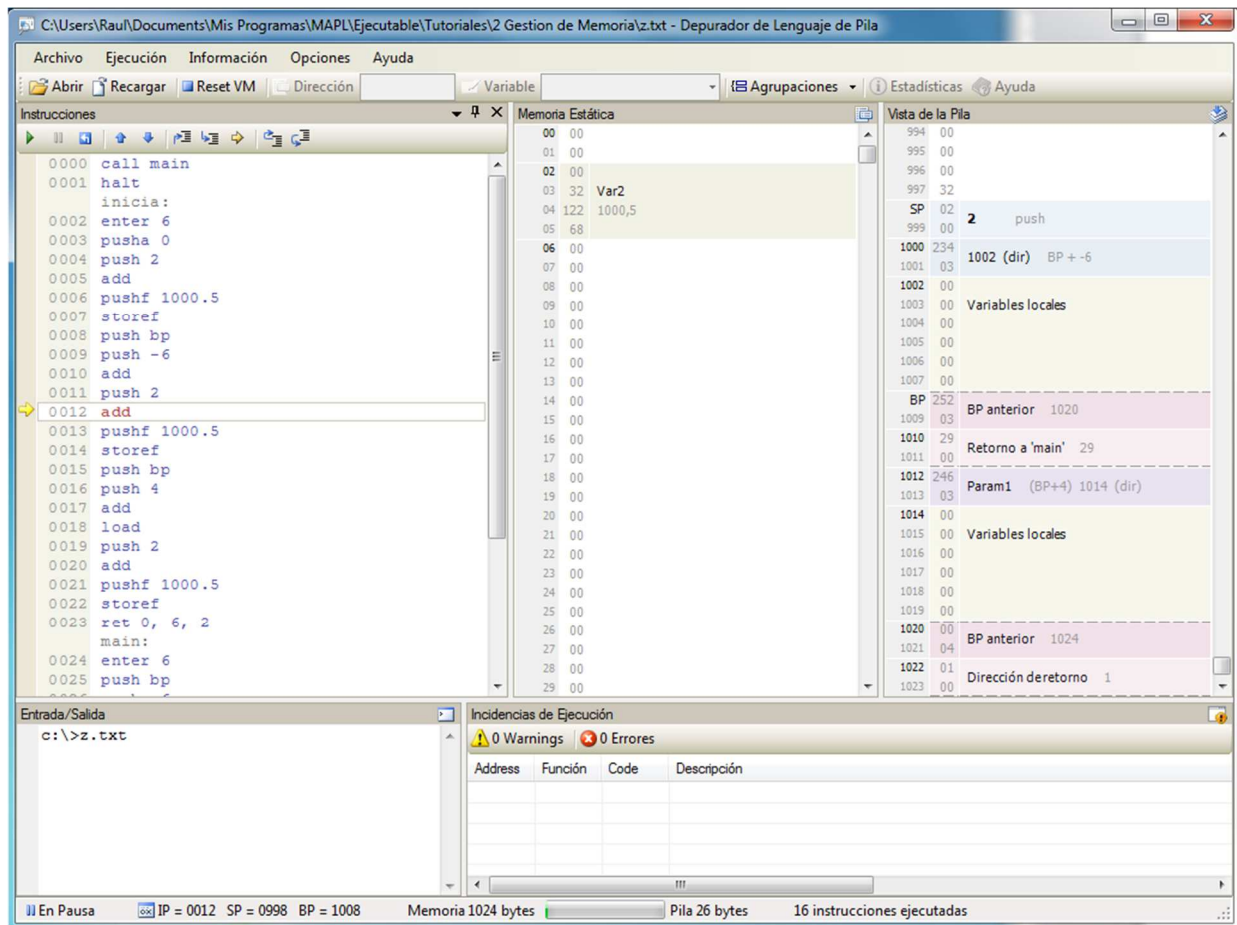


Ilustración 9. Depuración sin fichero de alto nivel

3 Tutorial

3.1 Estructura del tutorial

Para aprender el resto de las características de MAPL, lo más sencillo es seguir el tutorial ejecutando sus ejemplos en el propio GVM. Los ejemplos del tutorial están divididos en cuatro secciones (carpetas) que se recomienda seguir en orden.

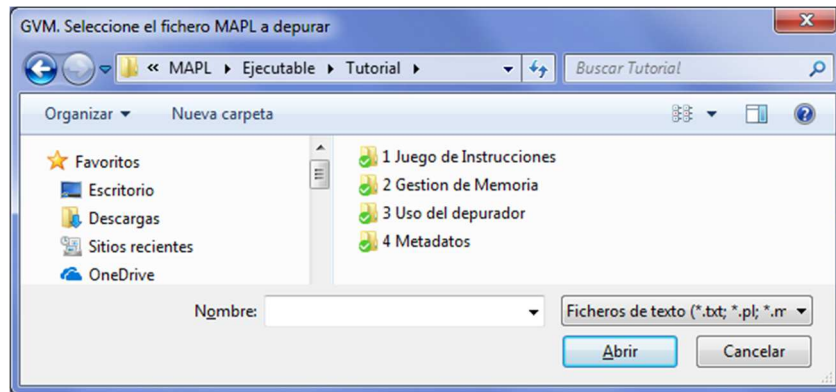


Ilustración 10. Las cuatro secciones del tutorial

Las dos primeras carpetas están pensadas para ser usadas como *soporte de la parte teórica*, mirando sus contenidos cuando en teoría se estén dando sus respectivos temas:

- **Juego de Instrucciones.** Explica de una manera visual el juego de instrucciones de MAPL.
- **Gestión de Memoria.** Muestra la ubicación de las variables en memoria en función de su ámbito (globales, locales y parámetros) y de su tipo (tipos simples, vectores y estructuras).

Las dos últimas carpetas están pensadas para ser realizadas *al comenzar la implementación* de la fase de generación de código del compilador.

- **Uso del depurador.** Muestra las opciones de las que se dispone para depurar un programa y así minimizar el tiempo para encontrar errores en el fichero generado.
- **Metadatos.** Muestra cómo usar los metadatos para declarar los símbolos de alto nivel (permitiendo así mostrar su disposición en memoria y aumentar las posibilidades del verificador) y cómo fusionar el programa generado con el fichero fuente a partir del cual se haya generado.

3.2 Ejecución de cada ejemplo

Los ejemplos están numerados dentro de cada carpeta. Solo hay que abrir los ficheros cuyo nombre comienza por un *número*. Los otros ficheros (sin numerar y acabados en ".Source.txt") son los fuentes de alto nivel de los ejercicios y *no* se deben abrir (GVM ya los abrirá y fusionará automáticamente cuando se abra el fichero que se corresponda con su bajo nivel).

Para seguir un ejercicio basta con ir ejecutando instrucción a instrucción (F7) e ir leyendo los comentarios que aparecen encima de cada bloque de instrucciones.

4 Metadatos

TextVM y GMV, además de las instrucciones de la máquina abstracta MAPL, admiten también directivas que, aunque son opcionales, son de mucha ayuda a la hora de comprobar si un fichero de salida tiene errores.

```
directiva → declaraciónGlobal
           | declaraciónDeFuncion
           | declaracionDeTipo
           | directivaDeFusion
           | directivaDeMemoria

declaraciónGlobal → #GLOBAL IDENT ':' tipo

declaraciónDeFuncion → #FUNC IDENT
                      | #RET tipo
                      | #RET VOID
                      | #PARAM IDENT ':' tipo
                      | #LOCAL IDENT ':' tipo

declaraciónDeTipo → #TYPE IDENT ':' tipo // Tipo de usuario

directivaDeFusion → #SOURCE \" IDENT \"
                   | #LINE LITERAL_ENTERO

directivaDeMemoria → #MEM LITERAL_ENTERO
```

Donde *tipo* se refiere a cualquiera de los siguientes:

```
tipo → INT
      | REAL | FLOAT // Sinónimos
      | BYTE | CHAR // Sinónimos
      | LITERAL_ENTERO '*' tipo // Array
      | '{' campos '}' // Struct
      | ADDRESS // Dirección o puntero
      | IDENT // Nombre de tipo de usuario

campos → campos campo
       | λ

campo → IDENT ':' tipo
```

En las siguientes secciones se mostrará brevemente el uso de cada una de las directivas. Sin embargo, es *muy* recomendable además abrir con GVM los programas que se encuentra en la carpeta "4 Metadatos" del tutorial de MAPL para ver más ejemplos de estas directivas y comprobar los efectos que se producen al modificarlos.

4.1 Directivas para la declaración de variables globales

Con *#global* se puede indicar la posición y el nombre de las variables globales.

```
declaraciónGlobal → #GLOBAL IDENT ':' tipo
```

Puede usarse *#var* o *#data* en lugar de *#global* (son sinónimos) y es indiferente que aparezcan en mayúsculas o minúsculas.

Ejemplo de entrada	Salida	Resultado obtenido en GVM																																																																												
int edad; char inicial; float sueldo;	#global edad:int #global inicial:byte #global sueldo:real	<table><tr><th colspan="2">Memoria Estática</th></tr><tr><td>00</td><td>00</td></tr><tr><td>01</td><td>00</td></tr><tr><td>02</td><td>00</td></tr><tr><td>03</td><td>00</td></tr><tr><td>04</td><td>00</td></tr><tr><td>05</td><td>00</td></tr><tr><td>06</td><td>00</td></tr><tr><td>07</td><td>00</td></tr><tr><td>08</td><td>00</td></tr><tr><td>09</td><td>00</td></tr><tr><td>10</td><td>00</td></tr><tr><td>11</td><td>00</td></tr><tr><td>12</td><td>00</td></tr><tr><td>13</td><td>00</td></tr><tr><td>14</td><td>00</td></tr><tr><td>15</td><td>00</td></tr><tr><td>16</td><td>00</td></tr><tr><td>17</td><td>00</td></tr><tr><td>18</td><td>00</td></tr><tr><td>19</td><td>00</td></tr><tr><td>20</td><td>00</td></tr><tr><td>21</td><td>00</td></tr><tr><td>22</td><td>00</td></tr><tr><td>23</td><td>00</td></tr><tr><td>24</td><td>00</td></tr><tr><td>25</td><td>00</td></tr><tr><td>26</td><td>00</td></tr><tr><td>27</td><td>00</td></tr><tr><td>28</td><td>00</td></tr><tr><td>29</td><td>00</td></tr><tr><td>30</td><td>00</td></tr><tr><td>31</td><td>00</td></tr><tr><td>32</td><td>00</td></tr><tr><td>33</td><td>00</td></tr><tr><td>34</td><td>00</td></tr><tr><td>35</td><td>00</td></tr><tr><td>36</td><td>00</td></tr></table>	Memoria Estática		00	00	01	00	02	00	03	00	04	00	05	00	06	00	07	00	08	00	09	00	10	00	11	00	12	00	13	00	14	00	15	00	16	00	17	00	18	00	19	00	20	00	21	00	22	00	23	00	24	00	25	00	26	00	27	00	28	00	29	00	30	00	31	00	32	00	33	00	34	00	35	00	36	00
Memoria Estática																																																																														
00	00																																																																													
01	00																																																																													
02	00																																																																													
03	00																																																																													
04	00																																																																													
05	00																																																																													
06	00																																																																													
07	00																																																																													
08	00																																																																													
09	00																																																																													
10	00																																																																													
11	00																																																																													
12	00																																																																													
13	00																																																																													
14	00																																																																													
15	00																																																																													
16	00																																																																													
17	00																																																																													
18	00																																																																													
19	00																																																																													
20	00																																																																													
21	00																																																																													
22	00																																																																													
23	00																																																																													
24	00																																																																													
25	00																																																																													
26	00																																																																													
27	00																																																																													
28	00																																																																													
29	00																																																																													
30	00																																																																													
31	00																																																																													
32	00																																																																													
33	00																																																																													
34	00																																																																													
35	00																																																																													
36	00																																																																													
int cuatrimestre[3];	#global cuatrimestre:3 * int	edad (sin inicializar)																																																																												
struct Punto { x:int; y:int; };	#type Punto: { x:int y:int }	inicial (sin inicializar)																																																																												
Punto centro;	#global centro:Punto	sueldo (sin inicializar)																																																																												
int *puntero;	#global puntero:address	cuatrimestre[0] (sin inicializar)																																																																												
float muestras[10][20];	#global muestras:10 * 20 * real	cuatrimestre[1] (sin inicializar)																																																																												
		cuatrimestre[2] (sin inicializar)																																																																												
		centro.x (sin inicializar)																																																																												
		centro.y (sin inicializar)																																																																												
		puntero (sin inicializar)																																																																												
		muestras[0][0] (sin inicializar)																																																																												
		muestras[0][1] (sin inicializar)																																																																												
		muestras[0][2] (sin inicializar)																																																																												
		muestras[0][3] (sin inicializar)																																																																												
		muestras[0][4]																																																																												

Con la información proporcionada por *#global*, además de poder mostrar la ubicación de las variables en memoria (columna derecha de la tabla anterior), el verificador podrá detectar accesos a memoria inválidos por no coincidir con el tipo de la variable o bien intentar acceder a una de sus direcciones que no sea la de inicio.

4.2 Directivas para la declaración de Funciones

También hay directivas con las cuales se puede indicar el nombre, parámetros, variables locales y tipo de retorno de las funciones.

```

declaraciónDeFuncion → #FUNC IDENT
                        | #RET tipo
                        | #RET VOID
                        | #PARAM IDENT ':' tipo
                        | #LOCAL IDENT ':' tipo

```


Pueden colocarse en cualquier lugar del fichero (la declaración de cada función puede ir junto con su código - antes o después - o bien se puede juntar la información de todas las funciones).

Ejemplo de entrada	Salida	Resultado obtenido en GVM																																																			
<pre>int f(int a, float b) { int i; return 1; } main() { f(10, 20.2); }</pre>	<pre>0000 call main 0001 halt f: #func f #ret int #param a:int #param b:float 0002 enter 2 #local i:int 0003 push 1 0004 ret 2, 2, 6 main: #func main #ret void 0005 push 10 0006 pushf 20.2 0007 call f 0008 popi 0009 ret 0, 0, 0</pre>	<table border="1"> <tr> <td>1007</td> <td>00</td> <td></td> </tr> <tr> <td>SP</td> <td>00</td> <td></td> </tr> <tr> <td>1009</td> <td>00</td> <td>i (BP-2) (sin inicializar)</td> </tr> <tr> <td>BP</td> <td>252</td> <td></td> </tr> <tr> <td>1011</td> <td>03</td> <td>BP anterior 1020</td> </tr> <tr> <td>1012</td> <td>08</td> <td></td> </tr> <tr> <td>1013</td> <td>00</td> <td>Retorno a 'main' 8</td> </tr> <tr> <td>1014</td> <td>154</td> <td></td> </tr> <tr> <td>1015</td> <td>153</td> <td>b</td> </tr> <tr> <td>1016</td> <td>161</td> <td>(BP+4) 20,2</td> </tr> <tr> <td>1017</td> <td>65</td> <td></td> </tr> <tr> <td>1018</td> <td>10</td> <td>a (BP+8) 10</td> </tr> <tr> <td>1019</td> <td>00</td> <td></td> </tr> <tr> <td>1020</td> <td>00</td> <td></td> </tr> <tr> <td>1021</td> <td>04</td> <td>BP anterior 1024</td> </tr> <tr> <td>1022</td> <td>01</td> <td></td> </tr> <tr> <td>1023</td> <td>00</td> <td>Dirección deretorno 1</td> </tr> </table>	1007	00		SP	00		1009	00	i (BP-2) (sin inicializar)	BP	252		1011	03	BP anterior 1020	1012	08		1013	00	Retorno a 'main' 8	1014	154		1015	153	b	1016	161	(BP+4) 20,2	1017	65		1018	10	a (BP+8) 10	1019	00		1020	00		1021	04	BP anterior 1024	1022	01		1023	00	Dirección deretorno 1
1007	00																																																				
SP	00																																																				
1009	00	i (BP-2) (sin inicializar)																																																			
BP	252																																																				
1011	03	BP anterior 1020																																																			
1012	08																																																				
1013	00	Retorno a 'main' 8																																																			
1014	154																																																				
1015	153	b																																																			
1016	161	(BP+4) 20,2																																																			
1017	65																																																				
1018	10	a (BP+8) 10																																																			
1019	00																																																				
1020	00																																																				
1021	04	BP anterior 1024																																																			
1022	01																																																				
1023	00	Dirección deretorno 1																																																			

Estas directivas presentan las siguientes ventajas:

- Nada más entrar en una función ya se muestra el nombre, tipo y dirección relativa (BP) de cada parámetro (en violeta de 1014 a 1019).
- Se muestran los nombres de las variables locales y su dirección relativa (dirección 1008).
- Al igual que ocurría con las variables globales, el verificador podrá detectar accesos inválidos a los parámetros o variables locales.
- Se comprobarán los operandos de la instrucción *ret* para que coincidan con los tamaños de las locales, parámetros y valor de retorno. Se comprobará además que la pila quede tal y como estaba al entrar en la función (que no haya quedado basura).

4.3 Declaración de Tipos de Usuario

MAPL permite la declaración de tipos de usuario. El uso fundamental de esta característica es la definición de estructuras.

```
#type Persona {
    edad:int
    dni:9 * char
}
#global pepe:Persona
#global juan:Persona
```

Sin embargo, puede usarse también para dar nombre a cualquier otra construcción de tipo:

```
#type miEntero:int
#global j:miEntero

#type empresa:10 * Persona
#global v1:empresa
#global v2:empresa
```

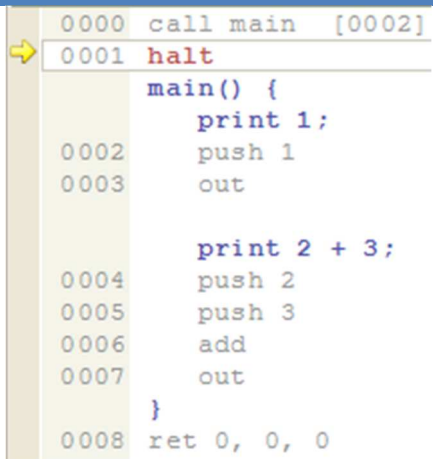

4.4 Directivas para la Fusión con alto nivel

Con `#source` y `#line` se puede asociar cada instrucción de MAPL con la línea del fichero fuente por la cual se ha generado. GVM usa esta información para presentar ambos ficheros entrelazados y facilitar así la revisión del código generado.

directivaDeFusion → `#SOURCE \“ IDENT \“`
| `#LINE LITERAL_ENTERO`

La directiva `#source` indica a partir de qué fichero de entrada se ha obtenido este fichero de salida, es decir, en qué fichero se encuentra el programa de alto nivel.

La directiva `#line` hace que el bloque de instrucciones MAPL que venga a continuación se dibuje debajo de la línea de alto nivel cuyo número se indica.

Fichero “entrada.txt”	Fichero “salida.txt”	Resultado obtenido en GVM
<pre>1. main() { 2. print 1; 3. 4. print 2 + 3; 5. }</pre>	<pre>#source "entrada.txt" call main halt main: #line 2 push 1 outi #line 4 push 2 push 3 addi outi #line 5 ret 0, 0, 0</pre>	 <pre>0000 call main [0002] 0001 halt main() { print 1; 0002 push 1 0003 out print 2 + 3; 0004 push 2 0005 push 3 0006 add 0007 out } 0008 ret 0, 0, 0</pre>

4.5 Tamaño de la memoria

Con la directiva `#mem` se puede indicar cuánta memoria (en bytes) se desea que tenga la máquina virtual para ejecutar el programa. Debe ser un numero entre 512 y 16384 (de medio Kb a 16Kb).

Si no aparece esta directiva (lo cual será lo habitual), el programa se ejecutará en una máquina virtual de 1 KB de memoria (lo cual es más que suficiente para cualquier ejercicio).

Ejemplo:

```
#mem 2048 // Se quiere ejecutar el programa sobre una máquina de 2 Kb de memoria
```

5 Arquitectura de MAPL

En esta sección se hará un rápido resumen de las características de MAPL. Para más detalles se debe seguir el tutorial con el propio *GVM*.

Secciones de memoria:

- Segmento de datos de 512 bytes a 16 Kb (1024 por defecto).
- Segmento de código en un espacio de direcciones separado de los datos en el que cada instrucción ocupa una dirección (comenzando en cero).

Distribución del segmento de datos:

- La memoria estática comienza en la dirección 0 del segmento de datos.
- La pila comienza en la última dirección del segmento de datos y crece hacia abajo (meter valores en la pila decrementa SP).

Registros:

- IP (segmento de código) Dirección de la instrucción actual.
- SP (segmento de datos) Dirección de la cima de la pila.
- BP (segmento de datos) Dirección del *stack frame* (dirección de retorno y antiguo BP) de la función actual.

Tamaño de los tipos primitivos:

- char = 1 byte
- int = 2 bytes
- float = 4 bytes
- dirección/puntero = 2 bytes

6 Apéndice. Juego de Instrucciones

Categoría	Bytes	Enteros (*)	Reales	Direcciones
Manipulación de la pila	pushb <i>cte</i>	pushi <i>cte</i>	pushf <i>cte</i>	pusha <i>cte</i>
	loadb	loadi	loadf	
	storeb	storei	storef	
	popb	popi	popf	
	dupb	dupi	dupf	
				pusha bp
Aritméticas		addi	addf	
		subi	subf	
		muli	mulf	
		divi	divf	
		mod		
Lógicas		and		
		or		
		not		
Comparación	>	gti	gtf	
	<	lti	ltf	
	>=	gei	gef	
	<=	lei	lef	
	==	eqi	eqf	
	!=	nei	nef	
E/S	inb	ini	inf	
	outb	outi	outf	
Conversiones (**)		i2b		
	b2i		f2i	
		i2f		

Categoría	Instrucción
Salto	jmp <i>label</i>
	jz <i>label</i> (<i>jump if zero</i>)
	jnz <i>label</i> (<i>jump if no zero</i>)
Funciones	call <i>label</i>
	ret <i>cte, cte, cte</i>
	enter <i>cte</i> / enter <i>-cte</i>
Otras	halt
	nop

(*) El sufijo *i* es opcional. Si no hay sufijo, se asume que es una instrucción para enteros:
push, load, add, gt, eq, in...

(**) La primera letra indica el tipo actual del valor y la última letra el tipo a convertirlo:
i2f → Convertir entero a float