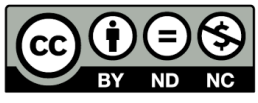
Máquina Abstracta MAPL

Contenido

Raúl Izquierdo Castanedo
raul@uniovi.es

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática

Escuela de Ingeniería Informática de Oviedo
Universidad de Oviedo



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 suma:

```
' - 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):

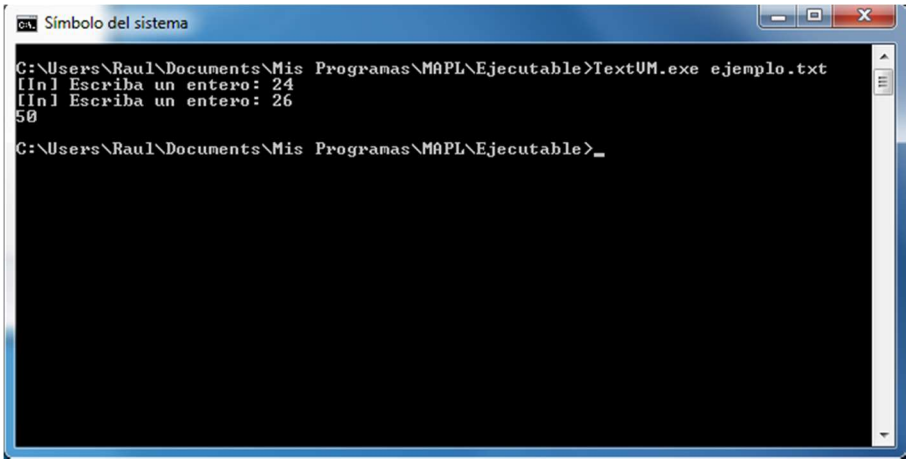


Ilustración 1. Ejecución con TextVM

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

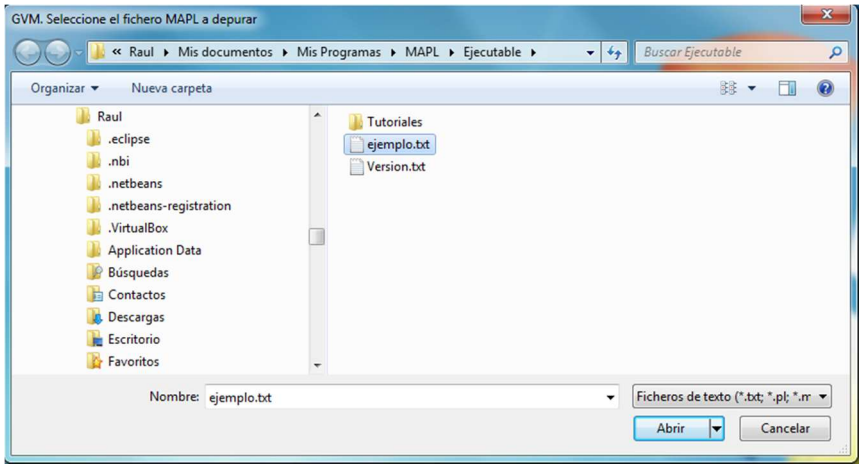


Ilustración 2. Seleccionar programa con GVM

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.

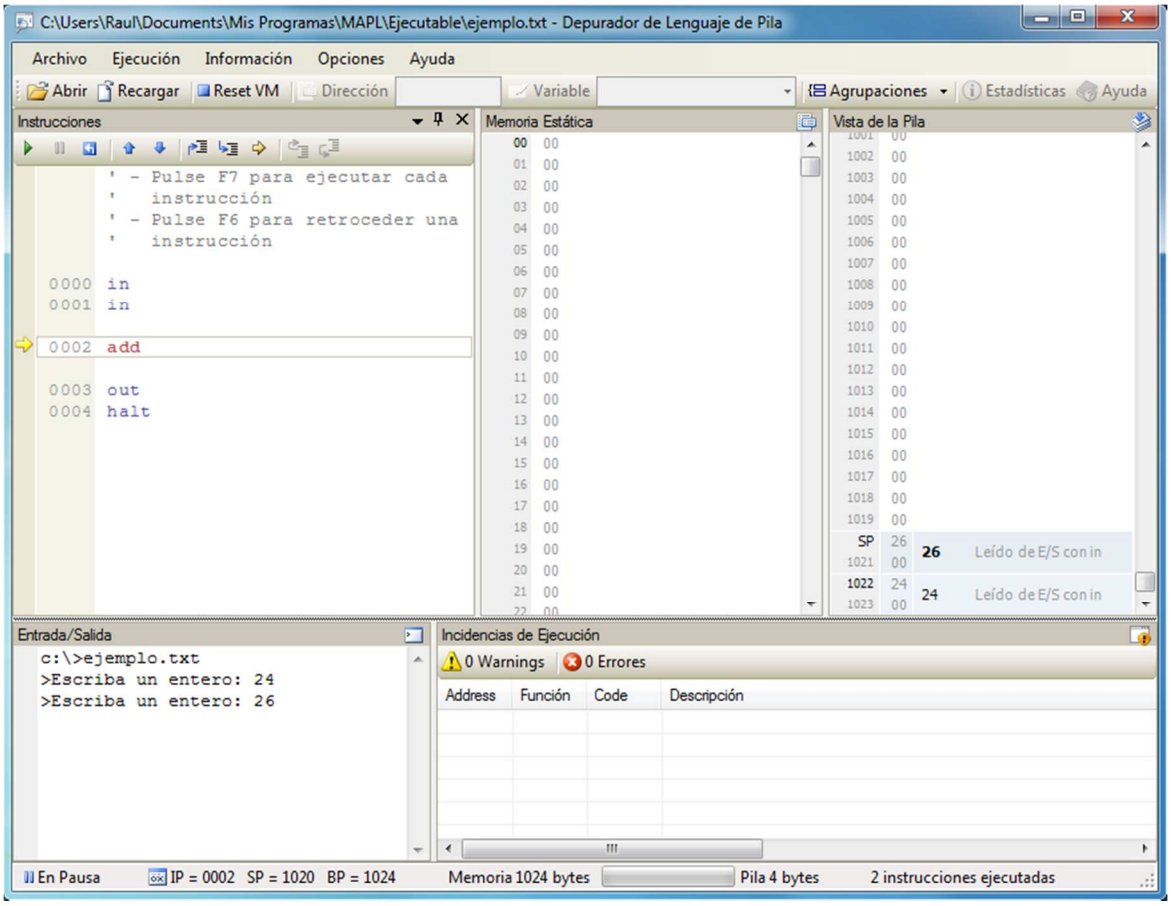


Ilustración 3. Ejecución paso a paso con GVM

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 MAPL[1]. Cada una de ellas se plantea como respuesta a uno de los dos objetivos establecidos anteriormente:

1. Funcionalidades para el objetivo de *enseñar*
   - Enseñar el juego de Instrucciones interactivamente
   - Mostrar la organización de la memoria dinámicamente
2. Funcionalidades para 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.

[1] El resto podrán verse en el tutorial

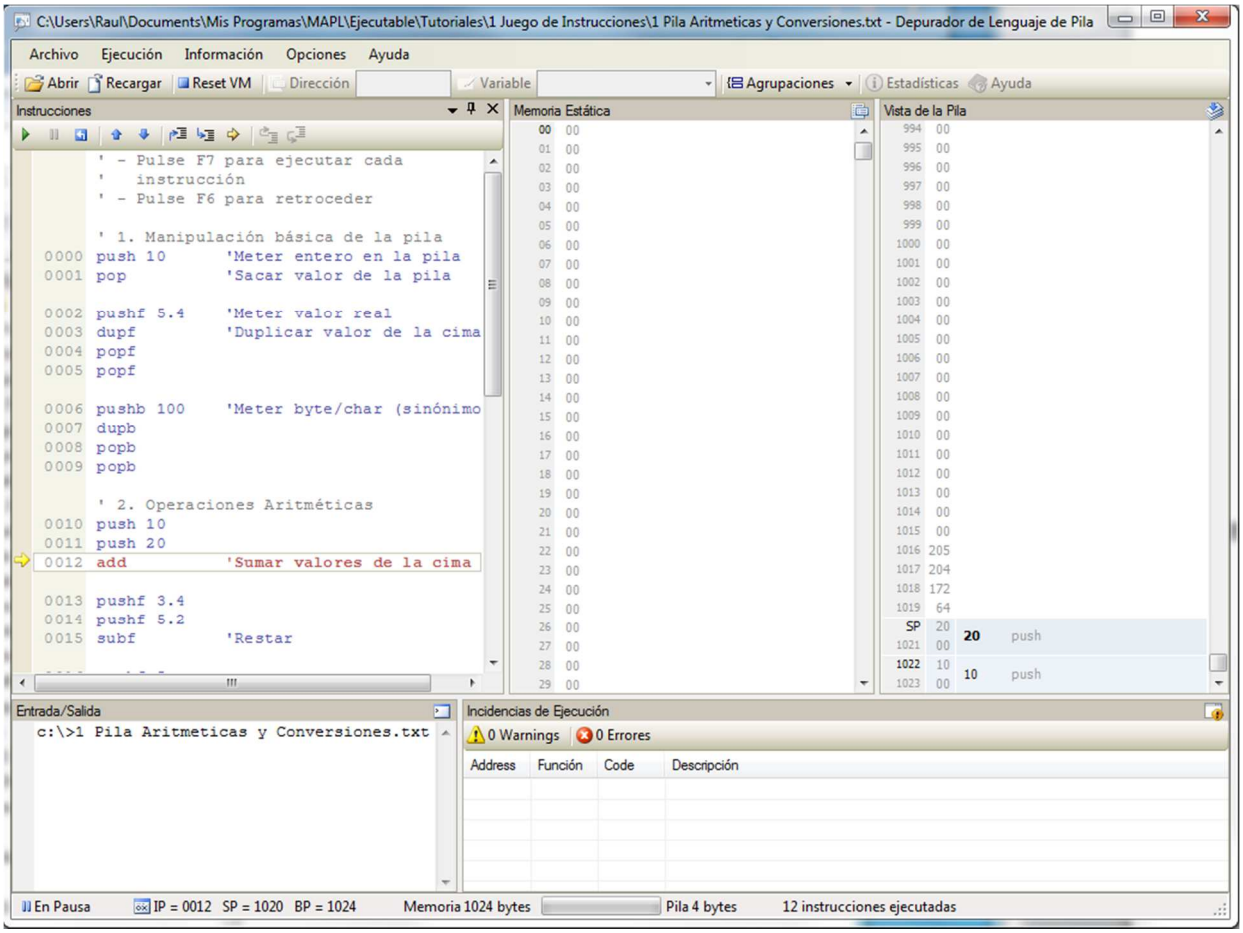


Ilustración 4. Ejercicio en el momento de mostrar la instrucción *add*

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 como 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.

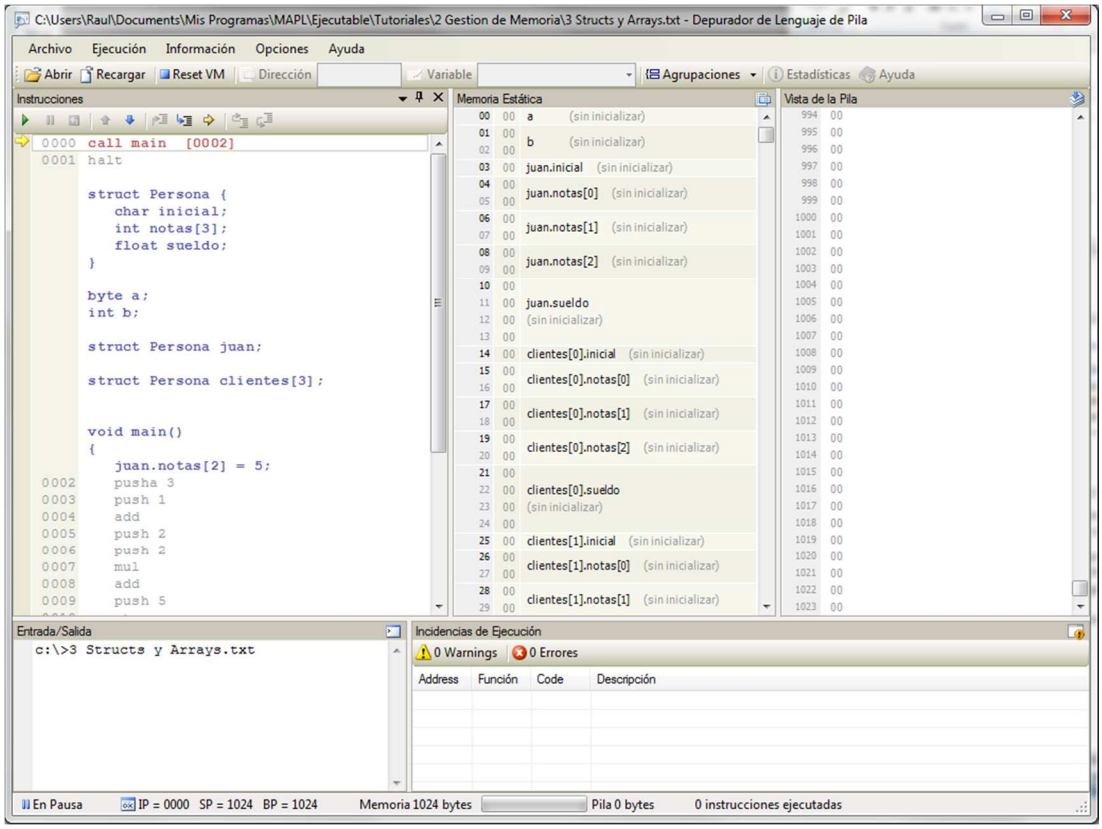


Ilustración 5. En el panel central (*Memoria Estática*) se muestra como se disponen las variables globales

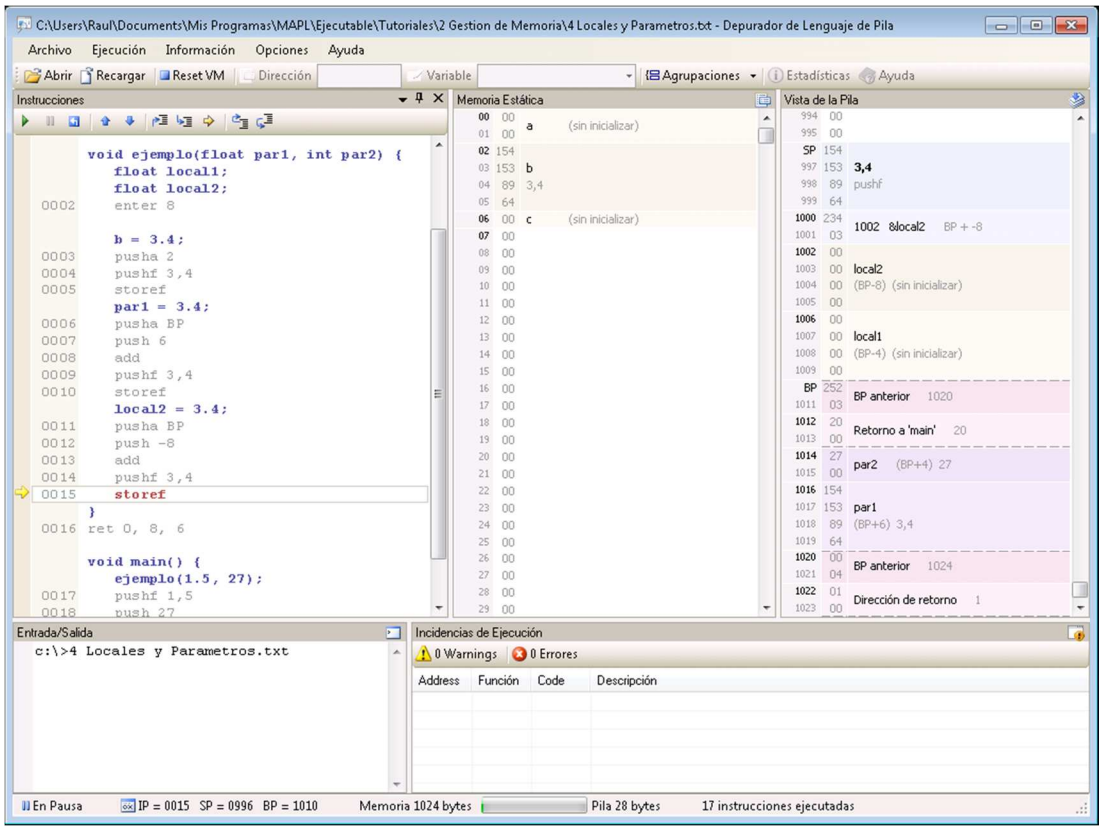


Ilustración 6. En el panel de la derecha (*Vista de Pila*) se muestran las variables locales y los parámetros junto con sus direcciones relativas (BP +/- constante)

2.3 Funcionalidades para *corregir*

MAPL ayuda a la verificación del compilador dando soporte directo a las tres tareas que hay que realizar durante esta fase:

1. Detectar lo más rápidamente posible *si hay errores* en la generación de código del compilador. Para ello *TextVM* y *GVM* incluyen verificación mediante comprobaciones semánticas.
2. Si los hay, saber el *por qué* el código generado es erróneo. Para ello *GVM* permite reproducir los errores.
3. Y una vez que se sepa por qué el código se ha generado mal, saber *qué parte del compilador hay que modificar* para generar el código correcto. Para 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 normalmente aparentemente sin errores y dejando la pila vacía. Sin embargo durante su ejecución se ha corrompido la pila[2].

```
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 el resultado del compilador* ha sido correcto (si ha generado las instrucciones adecuadas; cosa que no ha hecho).

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 basta con ejecutar el programa con alguno de ellos y observar si han encontrado situaciones anómalas[3].

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).
- Código muerto (no accesible). Posible síntoma de haber generado mal los saltos.

[2] Para saber más detalles de qué está mal solo hay que ejecutarlo con GVM y éste resaltará la línea del error y el por qué se produce.
[3] *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.

- 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 está tal y como estaba al entrar.
- Que los tamaños de variables locales, parámetros y tipos de retornos no sean coincidentes con los valores de los *ret* y el *enter* de cada función.

2.3.2 Reproducción de los errores

Una vez que se sabe que el código generado no está correcto lo que se necesita es saber el *por qué* 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:

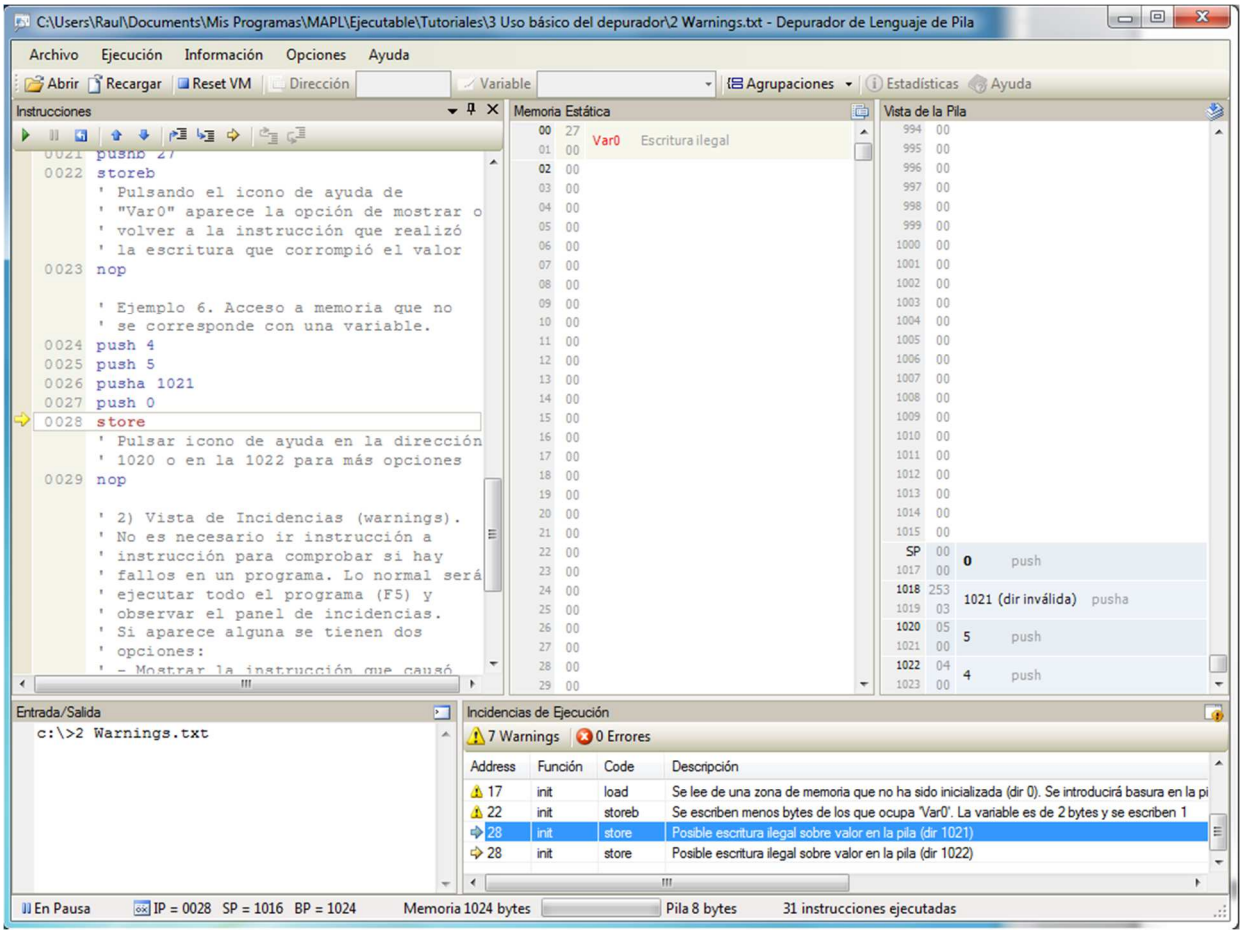


Ilustración 7. Warnings en panel inferior: rebobinando hasta el momento del tercer *warning*

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*.

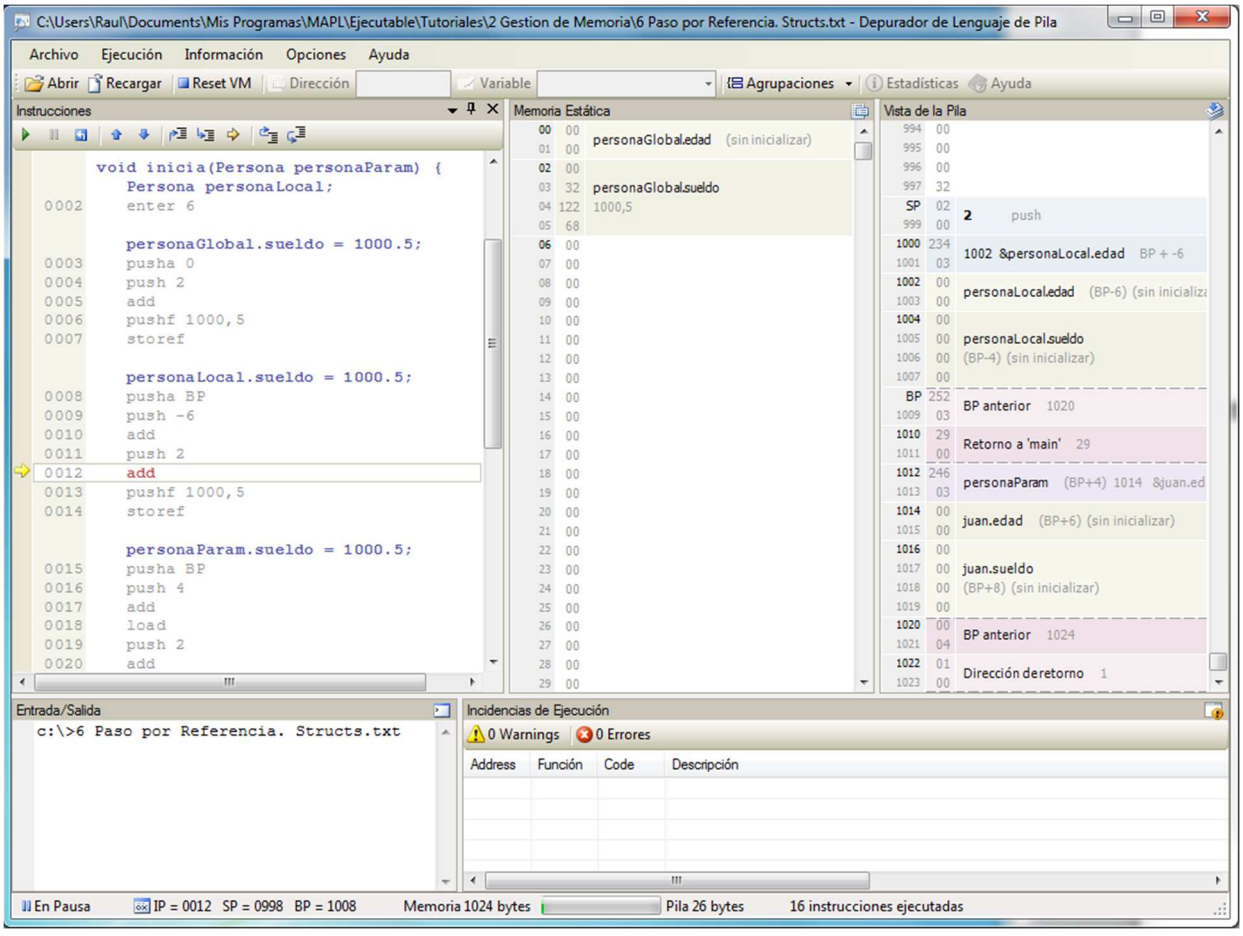


Ilustración 8. En el panel de la izquierda se muestra las instrucciones generadas por cada sentencia

Aunque la fusión con alto nivel es opcional, es altamente recomendable. Véase el mismo programa sin fusión de alto nivel. Es mas difícil saber a qué parte del compilador es la que ha generado la misma instrucción *add*.

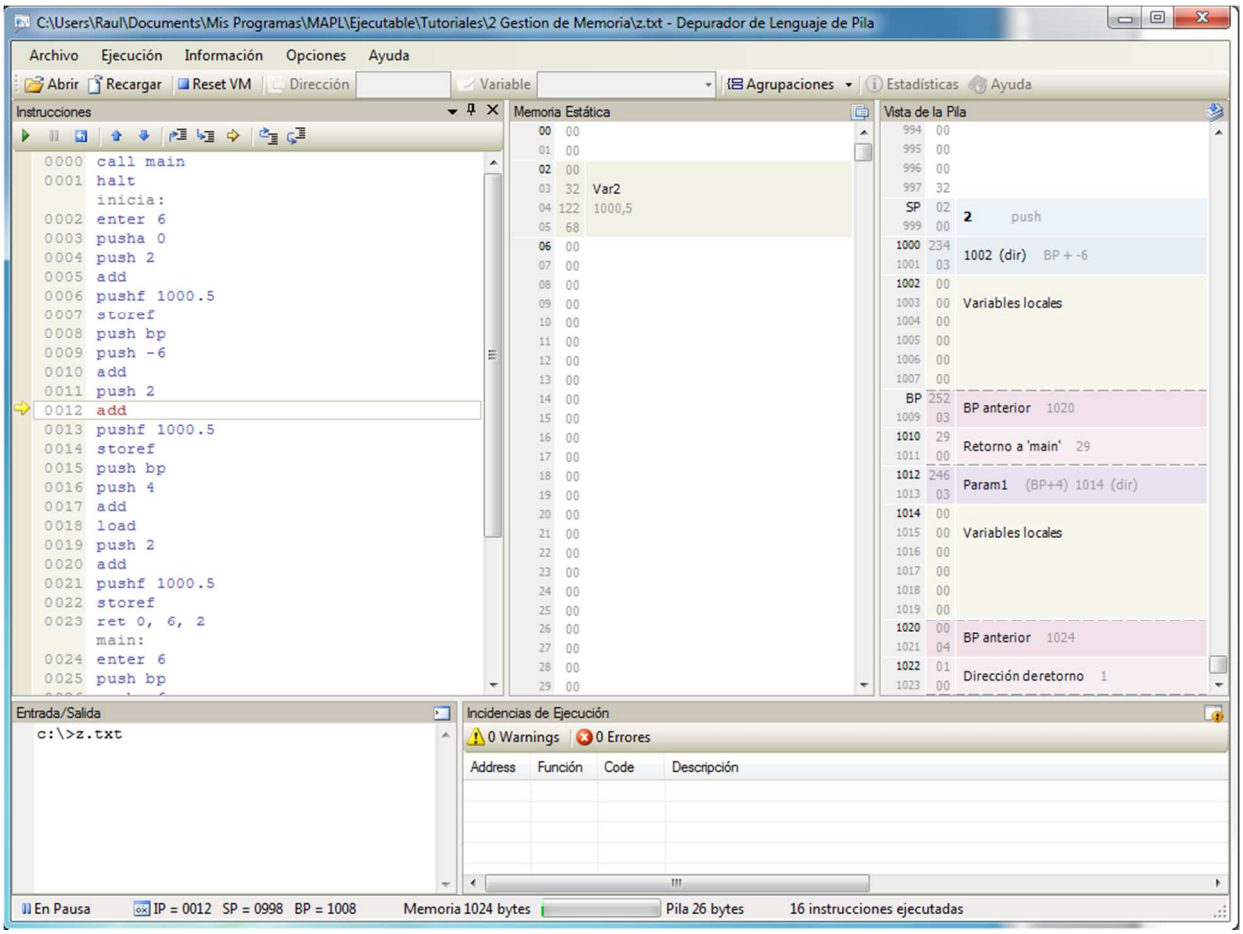


Ilustración 9. Depuración sin fichero de alto nivel

3 Arquitectura de MAPL

En esta sección se hará un rápido resumen de las características de MAPL. Para más detalles se debe seguir el tutorial de la arquitectura 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
- address/dirección/puntero = 2 bytes

4 Tutorial

4.1 Estructura del tutorial

Para aprender el resto de las características de MAPL lo más sencillo es seguir con GVM el Tutorial preparado a tal efecto. Los ejercicios del tutorial están divididos en cuatro secciones que se recomienda seguir en orden.

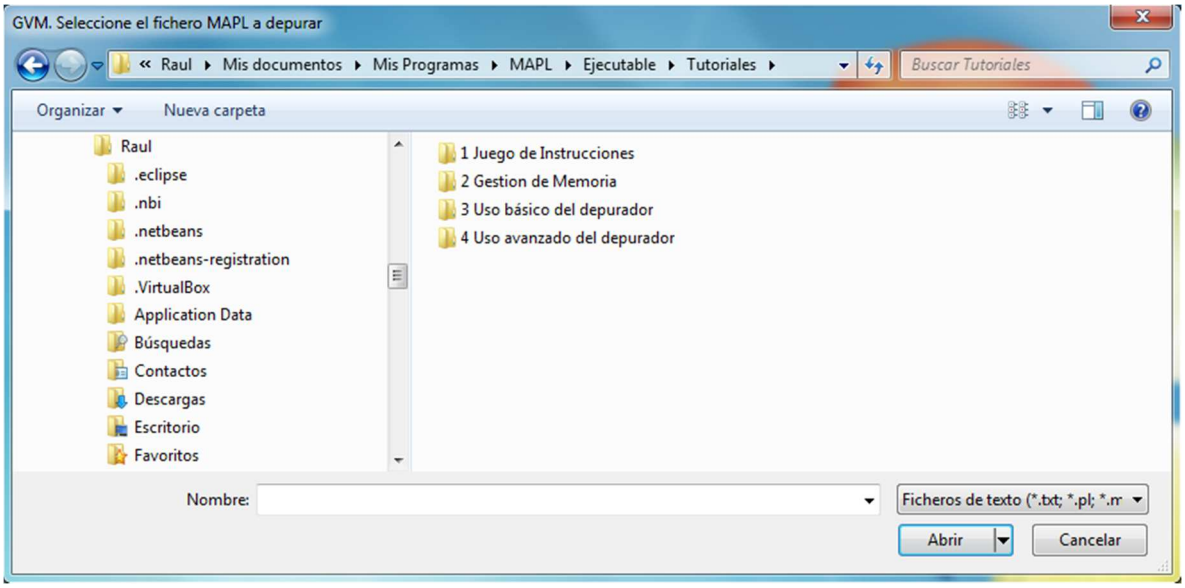


Ilustración 10. Secciones del tutorial

Las dos primeras carpetas están pensadas para ser usadas como *soporte de la parte teórica*, ayudando a entender de una manera visual el juego de instrucciones y la ubicación de las variables en memoria (variables locales, parámetros, 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. Tratan los siguientes temas:

- Cómo sacar provecho a la información y a las acciones que ofrece *GVM* para la depuración de los programas del alumno.
- Como usar los metadatos para mostrar la disposición de las variables en memoria.
- Como fusionar el programa generado con el fuente de con alto nivel.

Si el lenguaje del alumno no tiene funciones, bastará con que realice la carpeta "*3 Uso básico del depurador*". Si el lenguaje tiene funciones deberá realizarse además la carpeta "*4 Uso avanzado del depurador*".

4.2 Ejecución de cada ejercicio

Los ejercicios 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 los abrirá automáticamente).

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

5 Apéndice. Juego de Instrucciones

Juego de Instrucciones

Categoría	Bytes	Enteros (*)	Reales	Direcciones
Manipulación de la pila	pushb *cte*	pushi *cte*	pushf *cte*	pusha *cte*
	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	
Comparación <		lti	ltf	
Comparación >=		gei	gef	
Comparación <=		lei	lef	
Comparación ==		eqi	eqf	
Comparación !=		nei	nef	
E/S	inb	ini	inf	
	outb	outi	outf	
Conversiones (**)		i2b		
	b2i		f2i	
		i2f		

Categoría	Instrucción
Salto	jmp *label*
	jz *label* *(**j**ump if **z**ero)*
	jnz *label* *(**j**ump if **n**o **z**ero)*
Funciones	call *label*
	ret *cte, cte, cte*
	enter *cte*
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 del valor y la última el tipo a convertirlo:
i2f → Convertir entero a float

3