The MAPL Abstract Machine

Francisco Ortin
ortin@uniovi.es

Raúl Izquierdo
raul@uniovi.es

University of Oviedo, Computer Science Department, School of Computer Science Engineering.

1. Introduction

MAPL is an Abstract Machine for Programming Languages. It was designed as a target (intermediate) language for compiler implementation courses. MAPL is stack-based and most of its features are similar to the Java and .NET intermediate languages.

1.1. Implementation

MAPL is implemented as a .NET application, so you need the .NET Framework 2+ to run it. Windows operating system commonly provides a suitable .NET Framework. For Linux and Mac OS, you need to install any .NET Framework implementation such as .NET core or Mono.

MAPL can be run in two different ways:

- TextVM.exe (Text Virtual Machine). This file is a MAPL interpreter (pretty similar to java.exe). If you just want to run an assembly program in MAPL, the best choice is TextVM.
- GVM.exe (Graphical Virtual Machine). This visual application allows debugging MAPL assembly programs. It runs code step by step, showing the values of the different registers and changes in runtime memory. It is a powerful tool to detect errors in the MAPL code generated by a compiler (Section 1.3).

1.2. Implementation

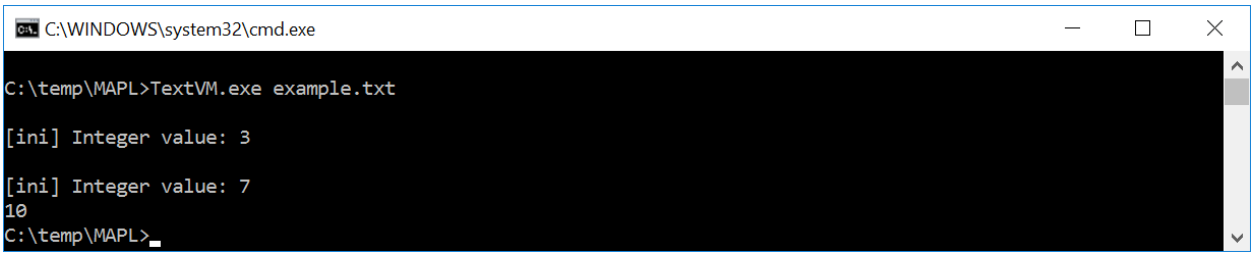
We can run the following example.txt file provided in MAPL:

```
ini
ini
addi
outi
```

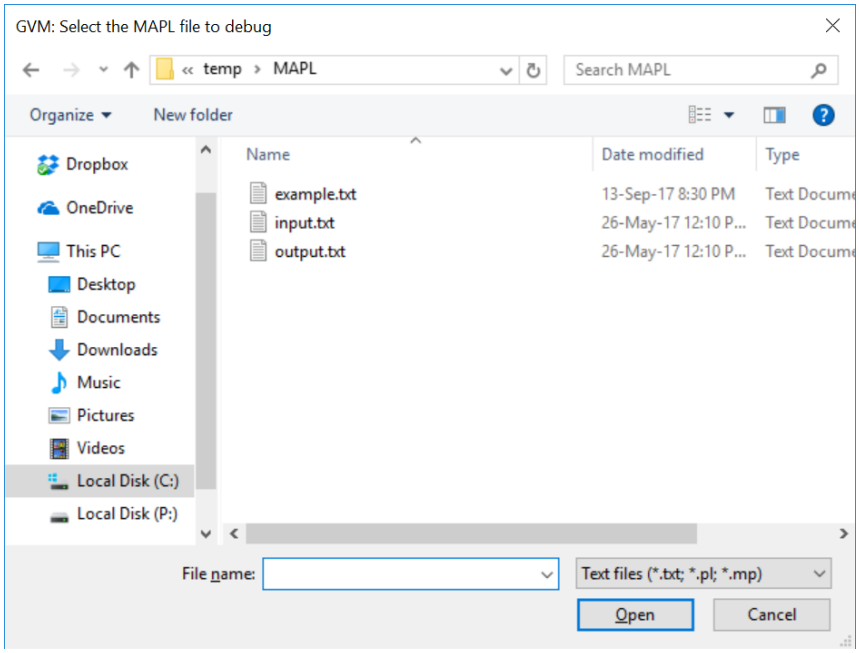
The code asks the user for two integers (INI = INput Integer), add them (addi = ADD Integer), and shows the result (outi = OUTput Integer).

We can run the program with TextVM the following way:

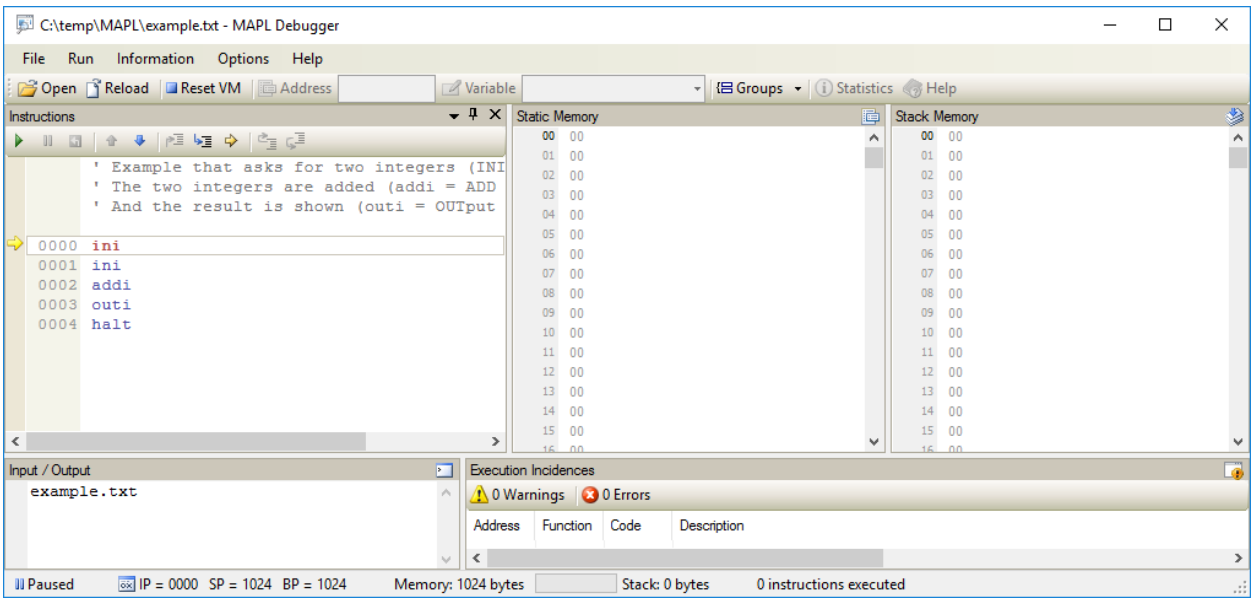
- Windows: Type `TextVM example.txt` in the command prompt (terminal).
- Mac OS and Linux: Type `mono TextVM.exe example.txt` in the terminal.



To run the same program with GVM, click on GVM.exe on Windows (or `mono GVM.exe` in Linux and Mac OS):



Then, select the program to be run (example.txt). The program is statically analyzed and, if no error is detected, it could be run / debugged:



We can press:

- F5 to run the whole program (as we did with TextVM).
- F7 to run one single instruction.
- F6 to undo the execution of the last instruction.
- F4 to run until the selected instruction. This functionality is the same as double-clicking on one instruction (runs the program until that instruction).

The rest of options are described in the IDE itself.

1.3. Benefits of MAPL

Even though MAPL is quite similar to Java and .NET assembly languages, there are some reasons to use MAPL in a programming language design and implementation course:

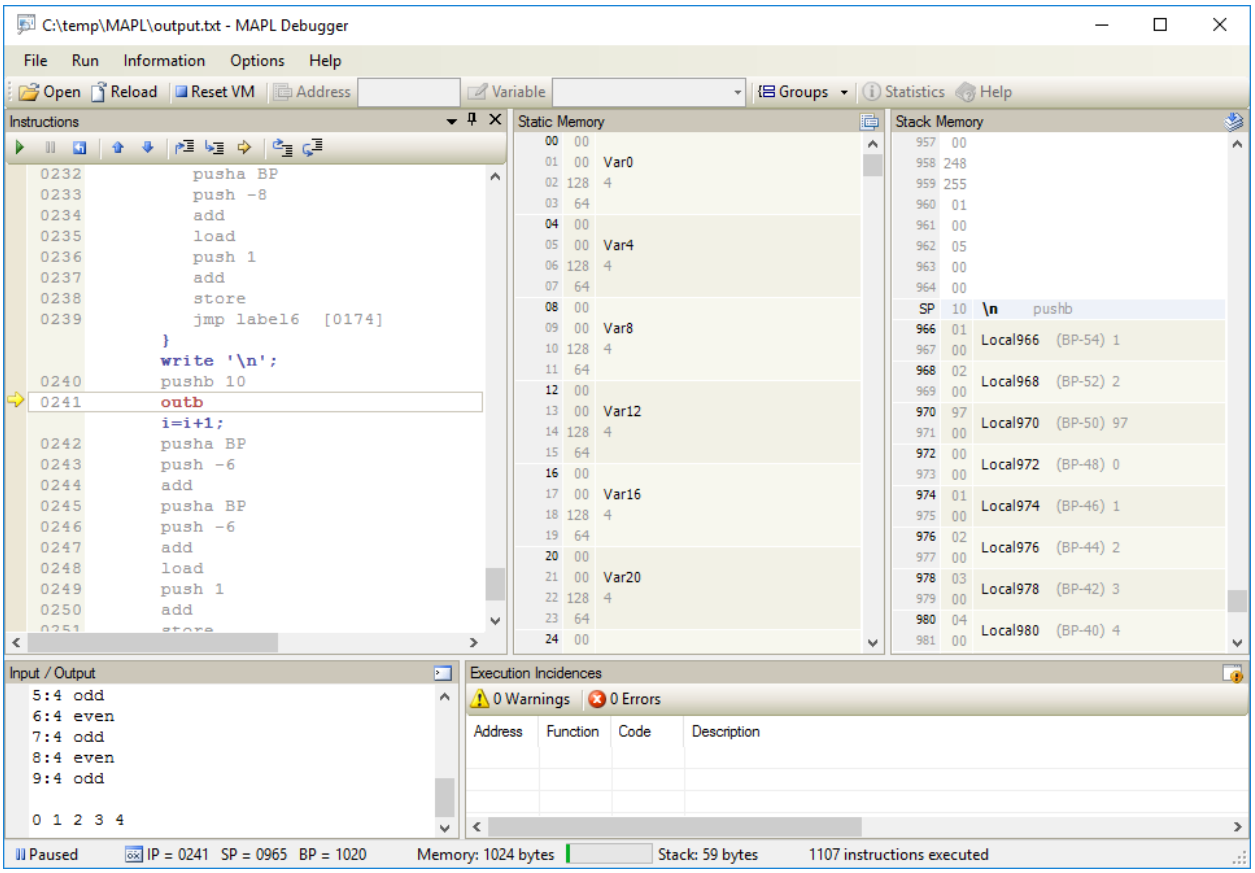
1) Error detection. MAPL GVM gives a lot of information at runtime, including a detailed explanation of what is causing the error. That makes it easier to detect the error in the generated code, and hence modify the compiler. Additionally, MAPL detects many common mistakes in assembly code that the existing VMs do not detect as erroneous (e.g., dead code, access to uninitialized memory and functions that do not return the execution flow to the caller).

2) Learning purposes. GVM provides very much information about the execution environment (e.g., registers, memory, local variables, parameters and the last instruction modifying each variable). This information is useful to understand how virtual machines work and to infer the variable addressing and runtime environment a compiler must define for the generated code. Additionally, GVM merges the assembly low-level instructions with the source high-level code, providing information about how the compiler translated one high-level statement into a collection of low-level assembly instructions.

3) Evaluation. Lectures can use MAPL to detect if the generated code has any static or dynamic error, caused by mistakes in the language implementation. Information given by runtime errors are very helpful to guess their cause.

2. GVM

The GVM application has the following structure:



The left panel (Instructions) shows the application execution trace; current instruction is highlighted and pointed with a yellow arrow. Low-level and high-level code is shown in gray and blue, respectively.

The center and right panels show the runtime state of static (global variables) and stack memory (local variables and parameters); MAPL provides no heap memory. Static memory grows downward (growing memory addresses) and stack memory grows upward. Actually, both panels show the same information: static memory presents the lowest memory addresses and stack shows the last ones. For each variable, MAPL displays its internal identifier (Var0, Var1…), value, memory address and the value of each byte comprising the variable.

Bottom-left panel shows the input and output messages to interact with the user (the console). Bottom-right panel presents the execution incidences occurred. Warnings allow program execution, but warn that a runtime error might have occurred (e.g., reading an uninitialized memory address). Errors stop program execution because something went wrong (e.g., adding two integers when there is not data on the top of the stack). Double-clicking the on the error makes MAPL to rewind the execution state to the state causing that incidence. In this way, we can easily detect the source of the runtime error.

The very bottom of the window presents the execution state (e.g., Paused), the values of the machine registers (IP, SP and BP), the size of memory being used and the number of instructions executed.

3. Architecture of the MAPL virtual machine

MAPL is a 2-byte word stack-based virtual machine. Memory is divided in two segments:

1) Memory segment (from 512 bytes to 16KBs). By default, 1024 are assigned. If more memory is required, the new memory size can be set with the #memory directive in the assembly file (e.g., `#memory 2048`). As mentioned, static memory starts in address 0 and grows upward; and stack memory starts in the last address, growing downward. MAPL follows a little-endian approach, storing least significant bytes first.

2) Code segment. Each instruction is placed in a growing memory address. All the instructions have a fixed size of one byte.

MAPL provides three 2-byte (one word) registers:

1) IP (Instruction Pointer) holds the value of the current instruction (the one to be executed) in the code segment.

2) SP (Stack Pointer) saves the address of the top of the stack (memory segment). It points to the first byte of the last value pushed onto the stack.

3) BP (Base Pointer) points to the address of the active stack frame (memory segment). Precisely, it holds the memory address where the value of the previous BP register was stored.

Sizes of primitive types in MAPL are:

- Characters: 1 byte.
- Integers and memory addresses: 2 bytes.
- Float numbers: 4 bytes.

4. MAPL instruction set

Push instructions:

- `pushb <ASCII_code>` Pushes the character (1 byte) onto the stack.
- `push[i] <int_constant>` Pushes the integer literal (2 bytes) onto the stack.
- `pushf <real_constant>` Pushes the real number (4 bytes) onto the stack.
- `pusha <int_constant>` Pushes the integer address (2 bytes) onto the stack.
- `push[a] bp` Pushes the value of the bp register (2 bytes).

Load and store:

- `loadb`, `load[i]`, `loadf` Pop a memory address off the stack (2 bytes). Then, push onto the stack the content (1, 2 or 4 bytes) of the address popped in the previous step.
- `storeb`, `store[i]`, `storef` Pop from the stack 1, 2 or 4 bytes. Then, pop from the stack a memory address (2 bytes). The content of the memory address is replaced with the value popped in the first step.

Popping and duplicating values on the stack:

- `popb`, `pop`[`i`], `popf` Pop 1, 2 o 4 bytes, respectively, off the stack.
- `dupb`, `dup`[`i`], `dupf` Duplicate the 1, 2 or 4 bytes, respectively, on the top of the stack.

Arithmetic operations: They pop two operands, perform the operation and push the result.

- `add`[`i`], `addf` For addition.
- `sub`[`i`], `subf` For subtraction.
- `mul`[`i`], `mulf` For multiplication.
- `div`[`i`], `divf` For division.
- `mod`[`i`], `modf` For modulus.

Comparison operations. They pop two operands, perform the operation and push the result.

- `gt`[`i`], `gtf` For "greater than" comparison.
- `lt`[`i`], `ltf` For "lower than" comparison.
- `ge`[`i`], `gef` For "greater or equal" comparison.
- `le`[`i`], `lef` For "lower or equal than" comparison.
- `eq`[`i`], `eqf` For "equal to" comparison.
- `ne`[`i`], `nef` For "not equal" comparison.

Logical operations. Pop one or two integer operands, perform the operation and push the result.

- `and` For the "and" logical operation.
- `or` For the "or" logical operation.
- `not` For the unary "not" logical operation.

Input / Output:

- `outb`, `out`[`i`], `outf` Pop one value off the stack and shows it in the console.
- `inb`, `in`[`i`], `inf` Read a value from the keyboard and pushes it onto the stack.

Conversions:

- `b2i` Pops one character and pushes it as an integer.
- `i2f` Pops one integer and pushes it as a real number.
- `f2i` Pops one real number and pushes it as an integer.
- `i2b` Pops one integer and pushes it as a character.

Jumps:

- `<id>:` Defines one label for jumps and invocations (functions).
- `jmp <label>` Jumps (unconditionally) to the label specified as a parameter.
- `jz <label>` Pops one integer and jumps to the label if the popped integer is zero.
- `jnz <label>` Pops one integer and jumps to the label if the popped integer is not zero.

Functions

- `call <id>` Invokes the *<id>* function.
- `enter <int_constant>` Allocates *<int_constant>* bytes on the top of the stack.

- `ret <int_constant>, <int_constant>, <int_constant>` Returns from a function invocation. The first constant represents the bytes to return; the second one, the bytes of all the local variables; and the last one, the bytes of all the arguments.
- `halt` Terminates program execution.

Debugging info:

- `#source <string_constant>` Allows MAPL to associate assembly code to the high-level source program.
- `#line <INT_CONSTANT>` Allows MAPL to associate the assembly code corresponding to each high-level statement.

5. Example

Now it is time to play with MAPL and get fluent with its assembly language. You may modify and run the example.txt file provided. You can also analyze the input.txt (high-level C-- source program) and output.txt (a translation of input.txt to MAPL assembly language) to learn how the assembly instructions can be use to represent high-level structures such as arrays, structs and functions. Please, notice that the MAPL file to be run is output.txt (not input.txt).