# Status

## VM language - VMIL

We decided to design our own language, not only does that allow us to fit it to our Virtual Machine(VM) or vice versa, it also means we had a very limited amount of opcodes. So we have implemented and tested all of them. If we were to continue working on the VM more would likely be added so that we can support more features.

Currently our language, which we will describe later, is very simple. We need to write 4 lines of code just to add two numbers. This, while not limiting the power of the language, seriously limits the productivity, it is very hard to code in and the number of lines needed for even simple programs is gigantic. You constantly have to make sure you don’t mess up the stack and sometimes you need to push object, such as the console, unto the stack way before you need it. So we really need are more high-level version of our language to abstract away all these difficulties.

We have also, because of limited time, implemented a lot of functionality into .NET and then used external functions to call them; we would have wished that we had time to implement more of it directly in our language.

The following are the opcodes comprising our language.

Opcodes

* load-field <field-offset>  
  Loads the field at the given offset onto the stack.
* store-field <field-offset>  
  Stores the value on top of the stack into the field at the specified offset.
* load-local <local-variable-offset>  
  Loads the local variable at the given offset onto the stack.
* store-local <local-variable offset>  
  Stores the value on top of the stack into the local variable at the given offset.
* push-literal <string-or-integer-literal>  
  Pushes the specified literal of type string or integer onto the stack. Internally this is converted to specific opcodes depending on the type of literal.
* pop  
  Discards the top stack element.
* dup   
  Duplicates the top element of the stack.
* new-instance  
  Expects to find the name of a class at the top of the stack which is then resolved into the actual class. A new instance of that class is then pushed onto the stack or a class not found exception is thrown.
* send-message  
  Sends the message located on top of the stack (in the form of a string) to the object found argument count elements further down the stack. Messages are always of the form <name>:<argument-count> and so the number of arguments can easily be found prior to message handler resolution – actually the number of arguments is part of the handler name thus allowing overloading.
* return-void  
  Returns with no value.
* return  
  Returns the value at the top of the stack.
* <label>:  
  Specifies a new label.
* jump <label>  
  Uncontional jump to label.
* jump-if-true <label>  
  Sends the message is-true:0 to the object at the top of the stack and jumps depending on the values returned (greater than 0 is true, everything else is false). Null at top of stack implies false.
* jump-if-false <label>  
  As above but the message sent is is-false:0 and null at the top of stack implies true.
* throw   
  Throws the exception at the top of the stack.
* .try { <instructions> } catch { <instructions> }  
  Normal try-catch-clause. Unlike the Java or .NET VM we recorded entry into try-catch block with special stack values.

Follwing are two examples of programs written in our language. First we simply add two numbers.

push-literal 1

push-literal 1

push-literal “add:1” //(1).add(1)

send-message

A slightly more complex example is getting an element from an array and printing it. We have stored the console instance in a field called con and the array in a field called arr. The object representing the console is stored in a variable as we have no static members.

load-field con

load-field arr

push-literal 1

push-literal “get:1”

send-message

push-literal “write-line:1”

send-message

In C# this would be something like: System.Console.WriteLine(arr[1]).

The syntax of the language is described in the following in an informal grammar. A precise grammar for the language can be found in the source code.

<class-declaration> =

.class <visibility> <name> extends <full-class-names> {

.fields { <names> }

.handler <visibility> <name>( <names> ) {

.locals { <names> }

<instructions>

}

.handler <visibility> <name>

.external <external-name>( <names> )

.default {

.locals { <names> }

<instructions>

}

<class-declarations>

}

<visibility> = public | protected | private

<name> = normal C style identifier

<full-class-name> = <name>(.<name>)\* // Full path to class

<external-name> = <name>(.<name>)\* // Path to method in SystemCalls

Since we have a very limited opcode set many common functions such as integer addition and subtraction are external library functions. This limits the number of functions we can implement directly in our language and also impacts performance. Since so many of the basic functions are external, we often need to switch from the interpreter to a .NET method. Currently that probably does not hurt performance that much but if we wanted to do JIT compilation and inline-caching it likely would since these methods cannot easily be inlined. For many of the most basic operations it would be beneficial if we could inline them in the interpreter – like recognizing the message add:1 if two integers are found on the stack.

The currently implemented API is actually quite large considering the nature of the project and a listing can be found in [REF].

## Status compared to initial plan

Initially we laid out six milestones, with the intention of completing one per week. This turned out to be very unrealistic, since none of us had implemented a VM before. Even after one week it was clear that the time table was slipping. We did however manage to complete the first three milestones. The last three which we described as “nice-to-have and not must-have” were only partly completed.

### Milestones 1, 2 & 3

M1 said that we should be able to execute most of our language and M2 said to support the full language and a directly loadable binary format. We initially had a binary format but dropped it later on, however we do support the full specification of our language. M3 said simple stop-and-copy GC. We did manage to implement a working GC but we decided to do mark-and-sweep instead of a copying-collector.

### Milestone 4

In M4 we had planned to do a generational collector and while we did implement a GC it is not generational. It would be interesting to see how our VM perform under different GCs but we ran out of time and never managed to make that comparison.

### Milestone 5

As M5 we had planed “JIT-compiler, may not include inline-caching”. We never got around to looking into JIT-compilation but it would without doubt have helped with performance, the same goes for inline-caching.

### Milestone 6

The final milestone we had planed “.NET interoperability, threading, inline-caching, trace-based compilation, compiling from gbeta”. We did manage to include full support for threading, but neither inline-caching, trace-based compilation, compiling from gbeta or .NET interoperability where possible within the time frame.

### Not in initial plan

Besides the things we planned to implement according to the plan, we also managed to implement a neat feature that allows us to change interpreter at run time, meaning we can swap in a debug interpreter mid program. We did unfortunately not manage to finish the debugging framework.