Bytecode Programming Language

Paolo Bettelini

Contents

1 Introduction

We have developed a custom programming language to design smart contracts, *piccions*.
The following sections contain a generic approach to designing a bytecode and the final implementation itself.

2 Bytecode

The bytecode is a set of low-level 1-byte long instructions (opcodes) in which a program is compiled. This set of instruction is run on a Virtual Machine (VM).

The life of a program looks as follows:

Source code → compiler → bytecode → VM

When the program is run on the Virtual Machine it is given a finite amount of memory (RAM), this memory is divided into stack and heap.

Programs run in a virtual machine are generally slower, but the advantage is that the compiled program is not CPU-specific and the program can be safely virtualed within the VM itself.

3 Stack

The stack is a piece of memory in which values are stacked on top of eachother. The purpose of the stack is to keep in memory temporary values for evaluating nested expressions and variables of the local scope.

4 Heap

The heap is the space in memory where all the objects are stored. The objects are (usually) indexed by 32-bit pointers which point to the beginning of the object within the heap. These pointers are the actual values stored in the stack. This means that for a given variable *a*, its pointers will be stored on the stack, and the pointer will point to *a* in the heap memory. The Java programming language however directly stored primitive values (numbers and such) on the stack, without a pointer to the heap.

5 Instructions

The two fundamental stack operations are **PUSH** and **POP**:
PUSH → pushes a value on top of the stack
POP → removes the topmost value from the stack

A simple bytecode for working with 8-bit values:

Code	**Stack before**	**Stack after**	**Description**
PSH		value	pushes the next value in the bytecode onto the stack
ADD	v1, v2	result	pop the two topmost values, adds them and pushes the result onto the stack
SUB	v1, v2	result	pop the two topmost values, subtracts them and pushes the result onto the stack
DIV	v1, v2	result	pop the two topmost values, divides them and pushes the result onto the stack
MUL	v1, v2	result	pop the two topmost values, multiplies them and pushes the result onto the stack

All of these operator work on and change the state of the stack.

Here is the code to compute $4 + 3$ and place the result on top of the stack

```
00   PSH    // push 4 onto the stack
01   4
02   PSH    // push 3 onto the stack
03   3
04   ADD
```

The stack now only contains the result of the addition, 8, which we could now print.
When the Virtual Machine will execute this code, it will do the following operations:

```
push(next())     // push next value
push(next())     // push next value
push(add(pop(), pop())) // pop two values, add them, push
```

Here's a simple pseudo-code for executing bytecode

```
while (!done) {
    var code = next()

    if (code == PSH)
        var value = next()
        push(value)

    if (code == ADD)
        var v1 = pop()
        var v2 = pop()
        var result = v1 + v2
        push(result)

    if (code == MUL)
        var v1 = pop()
        var v2 = pop()
        var result = v1 * v2
        push(result)

    if (code == SUB)
        var v1 = pop()
        var v2 = pop()
        var result = v2 - v1
        push(result)

    if (code == DIV)
        var v1 = pop()
        var v2 = pop()
        var result = v2 / v1
        push(result)
}
```

You might notice that in the DIV and SUB code, we compute v2 / v1 or v2 - v1 instead of using v1 and then v2. This is because pushing elements on top of the stack means that they will be popped in the reverse order, for MUL and ADD this is not a problem since they are commutative operations.

6 Nested expressions

You might think that we need some extra steps, however our stack already supports all kinds of complex nested computations.
Let's consider the following nested expression: **(2 + 2) * 4 / ((3 - 1) * 2)**.

- Start with 2+2
- Multiply it by 4

At this point we must "pause' the current value to compute to other side of the expression

- Compute 3-1
- Multiply by 2

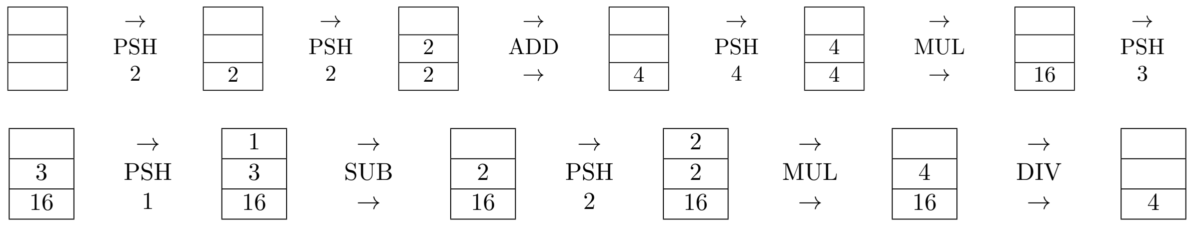
And now we will have both values on top of the stack

- Divide

The bytecode looks as follows

```
    PSH 2
    PSH 2
    ADD
    PSH 4
    MUL
    PSH 3
    PSH 1
    SUB
    PSH 2
    MUL
    DIV
```

Here's how the stack evolves whilst computing the expression



The beauty of the stack is that it will keep stacking elements on top of eachother, but eventually everything will simplify and the result will be on top of the stack.

With the stack, you can compute any nested expression no matter how complex. The only limitation is the size of the stack, if an element is pushes to the stack but there's no more space for it, the program crashes. This is called a stack overflow.

7 Data types

So far we've only worked with 8-bit (1 byte) values. This means that every value pushes or popped from the stack is very limited in size [-127;128] or [0;255] depending on how you do the math on the Virtual Machine. Eventually you might want to define other data types, such as 32-bit integers (4 bytes). When pushing an i32 onto the stack, you will need to decompose it into its 4 bytes components, and push each one separately. The action of popping the i32 from the stack, is actually 4 different pops. Each byte will then be recomposed by the VM into an i32 so that you can work on it.

Another data type could be 32-bit floating-point numbers (f32), boolean values, strings and so on.

8 Storing variables

The stack is also used to store local variables (variables usable on the current scope of the code). This is done by using a stack frame, which is a frame od data that gets pushes onto the stack.
Whenever a function is called (we'll cover functions later, for now we only have 1 big function which is the whole program) a stackframe is allocated. When the functions ends the stackframe is deallocated.

let's execute the collowing code

```
variable = 54
54 + 46
```

When compiling the code the compiler will decide how big the stack frame for each function will be, depending on the number of variables used. In this case we only need 1 variables (1 byte)

```
ALLOC    // Allocate stack frame of 1 byte
1
PSH      // Push 54
54
STORE    // Store 'variable' into the stack frame
−1
PSH      // Push 46
46
LOAD     // Load 'variable' from the stack frame
−2
ADD      // Add
```

STORE -1 This instruction can be decomposed as:

- pop a value from the stack
- store it at an offset (-1)

After the value is popped, the space for our variable within the stack frame is 1 byte before the current point, hence we use -1 as an offset.

When we call **LOAD -2**, the variable is 2 bytes from us since we previously pushes 46 onto the stack. The compiler automatically fills these values in

9 If statement

The first step to implement conditions is to define the bool (boolean) type. We only need 1-bit of information to store a boolean value, however we can only allocate a multiple of 8-bits.

We define the bool type with a byte:

$$\begin{aligned} 00000000 &\rightarrow \text{FALSE} \\ \text{else} &\rightarrow \text{TRUE} \end{aligned}$$

We still lack of a fundamental operation in the bytecode: **GOTO**
The **GOTO** operation tells the code to jump to another instruction.

To define the if structure, we could say that if the condition is FALSE, then we need to skip the body of the statement. For simplify sake, I'll define the **GOTO_IF_NOT** instruction, which pops a boolean value and goes to the instruction given by the next value in the bytecode if the bool value is false.
The condition boolean will be pushed on top of the stack before the if execution. It doesn't matter if it is hardcoded on the bytecode, read from the stack frame or is the return type of a function.

```
if (true) {
    PSH
    2
}
PSH
4
```

There are many ways to implement if statements, here's one:

```
PSH            // Push condition
TRUE
GOTO_IT_NOT    // Go to <checkpoint> if value is false
<checkpoint>
PSH            // If body
2
PSH            // Outer code (<checkpoint> instruction)
4
```

10 If-Else Statements

For the if-else statement we can just expand the if logic.
If the condition is not satisfied we will jump to the else body instructions.
If the condition is satisfied, we will execute the if body instructions. At the end of the if body we will jump after the else body.

```
- Condition on top of stack
- GOTO_IF_NOT <checkpoint1>
- ... (if body)
- GOTO <checkpoint2>
- ... (else body) <checkpoint1>
- ... (outer program) <checkpoint2>
```

Consider the following program:

```
 if (true) {
     PSH
     2
 } else {
     PSH
     4
 }
PSH 6
ADD
```

The bytecode looks as follows:

```
PSH
TRUE
GOTO_IT_NOT
<checkpoint1>
PSH
2
GOTO
<checkpoint2>
PSH                (<checkpoin1>)
4
PSH                (<checkpoin2>)
6
```

11 While loop

LOGIC:
Push condition
If the condition is not satisfied, jump to the end of the body
If the condition is satisfied, execute the body. At the end of the body jump to the beginning.

For this example a new instruction is needed: **EQUALS**

Code	**Stack before**	**Stack after**	**Description**
EQUALS	v1, v2	bool	Pops the two topmost values from the stack, compares them and pushes the (boolean) result onto the stack.

```
00  PSH  2
02  PSH  3
04  EQUALS
```

will result in FALSE in top of the stack

I am also going to use the **PRINT** (v1 -> _), which pops a values and "prints" it to some standard output.

Pushing a condition (from the bytecode) will result in either an infinite loop or no iterations at all. We will read and write to a variable in the stackframe, this is the pseudo-code:

```
variable = 10
while (variable / 10 == 2) {
    print(variable)
    variable = variable + 1
}
```

Here's the bytecode

```
    ALLOC 1        // Store variable=10
    PSH 10
    STORE −1

    LOAD −2        // Push condition <checkpoint1>
    PSH 10
    DIV
    PSH 1
    EQUALS

    GOTO_IF_NOT <checkpoint2>

    LOAD −1        // While body
    PRINT
    LOAD −1
    PUSH 1
    ADD
    STORE −1

    GOTO <checkpoint1>
    ... (outer program) <checkpoint2>
```

This will produce the following output (since integer division is rounded the loop will stop at *var = 20*):

```
10 11 12 13 14 15 16 17 18 19
```

12 Functions

The simpliest form of 'function' is a macro: the compiler places the same instructions multiple times throughout the program. This is not good memory-wise. We need a jump-system.

The first problem is that the function must be defined within the program but not executed (unless it has been called) A simple solution could be

```
    ... code

    GOTO <checkpoint1>
    ... (function body)

    ... code <checkpoint1>
```

When we want to call the function we can just jump to the beginning of its instruction. However, when the function has ended we must continue doing what we were previously executing. To solve this problem, before jumping to a function, we push the current position. When the function has finished, it pops the index and jumps to it.

```
    ... code

    GOTO <checkpoint1>
    ... (function body)      <function>
    GOTO <checkpoint2>
```

```
 ... code <checkpoint1>
GOTO <function>
 ... code <checkpoint2>
```

13 Function with parameters

Before jumping to the function, push the parameters values onto the stack. The function will pop them and use them.

14 Function with return value

Before the function finishes, it pushes the return value onto the stack. However, remember that the function must then jump back to the call checkpoint, but the topmost value is the return value rather than the index to jump to. To solve this we could create an instruction **SWAP**, which swaps the last two values on top of the stack. Using the **SWAP** operation, we can then pop the index to jump to. The following instructions will be able to pop the result value from the stack.

15 Storing objects and pointers

The solution is pointers and heap memory. The heap memory is another frame of memory (bigger than the stack) used to allocated every objects. Pointers are values (usually 32-bit integers) that point to an object within the heap memory. Instead of storing the *actual* values in the stackframe, we will only store pointers. Those pointers will point to the *actual* values we care about in the heap memory.

As stated at the beginning, Java doesn't use pointers for primitive values. Integers and such are directly stored in the stack frame (since they are as small as a pointer).

To explain how the heap memory works we can use the C language. The C programming language provides the malloc(size) function, which returns a pointer to a chunk of <size> in the heap that you can use. It also provides the free(pointer) method, to free memory from the heap.

```
void* chunk = malloc(100);
free(chunk);
```

Other languages don't require you to manually free memory:
Java implements a ”Garbage Collector”, which frees all the unused memory from the heap.
Rust automatically frees unused memory, however this features comes with some extra retrictions.