Bytecode interpreter

Fredrik Sørensen, Stein Krogdahl, Birger Møller-Pedersen

Spring semester 2021

Contents

1 Introduction

This report presents the local variant of a library for *bytecode construction and interpretation* written for the compiler construction course INF5110 at the University of Oslo. The objective of this exercise is to write a compiler for a simple example language. The bytecode library and intepreter were developed to be used in the *code generation* and *runtime* part of the obligatory exercises in the course.

Bytecode instructions are in general similar to instructions in machine code, but instead of being run directly on a machine, they are usually interpreted by a bytecode interpreter. Alternatively, they may also be translated into machine code before being run. A bytpecode instruction library is a library to simplify the task of generating bytecode for a bytecode interpreter. The terminology "bytecode" comes from the fact that the size of each instruction one single byte.

Our bytecode is quite similar to Java bytecode and our bytecode library is based on the Byte Code Engineering Library.[1] We choose to write a simpler, stripped-down version that, for instance, does not support classes, virtual procedures, and many other features supported by Java's bytecode. This way, it's simpler to work with and there would be less code. For instance, one does not need to create classes like, since our language does not have them, just for "some reasons unexplained". Still, we encourage students to look at BCEL, it's a nice and well designed tool

The bytecode library and the interpreter are, not surprisingly, written in Java. All the code is written in source form and consists of the following packages.

package	purpose
`bytecode.*`	classes to create bytecode
`bytecode.instructions.*`	instruction classes, supporting the above
`bytecode.type.*`	classes for types, supporting the above
`runtime.*`	classes for the runtime system

The code for the two packages `bytecode` and `runtime` is found in the corresponding two directories under `./src`

1.1 The interpreter

The interpreter is *stack based* and it interprets about 40 different instructions of our bytecode. The interpreter "automatically" handles allocation of struct types, method calls and access of variables in a struct when instructed to by the bytecode instructions. In the current version, there is **no garbage collector** and it just allocates memory sequentially as long as there is memory left.

The operators or instructions are so called **stack operations**. It means that whe the intpreter executes an instruction, it pops off a number of operands from the stack (0 or more), performs the task specified by the instruction, and leaves a number of result values on the stack, typically just one such value.

The interpreter is not written for efficiency, by rather for readability and ease of creating runnable bytecode. For instance, the types are kept on the stack together with the calculated

[1] `http://commons.apache.org/proper/commons-bcel/`, formerly as part of the Jakarta project.

values and the interpreter decides what kind of operation to perform based on the types as well as the current instruction. For example, an `ADD` instruction, representing addition, will be performed differently if there are two integers on the stack, two floats, or one of each. That's different from Java bytecode, which has an `ADDINT` and an `ADDFLOAT` instruction and where type casting has to be done explicitly in the bytecode (for example with the `i2f` instruction, which does the corresponding conversion).

1.2 Interpreting bytecode

As said, the interpreter is stack-based. We are not talking here about the run-time stack of the run-time environment, but the P-code like treatment of instruction. Of course, the interpreter must realize also some stack organizing calls and returns, scoping, and parameter passing.

The parameters of an an operation (including user-defined functions) must be placed on the stack with **the leftmost at the bottom and the rightmost on the top** of the stack before the instruction itself is interpreted. The top of the stack is the place where one pushes into and pops off. In the lecture, in the part of the run-time environments, the run-time stack was perhaps confusingly *depicted* with the top the stack at the bottom.

For example, the `SUB` instruction (for subtraction) requires that the two operands of the `SUB` instruction are on the stack in the correct order beforehand. The number to subtract must be on the top of the stack and the number to subtract from deeper in the stack. We may denote the elements on the stack before an instruction is interpreted with s_n, where n is the index from the top, with the top as $n = 0$. Then, the result of the `SUB` instruction is that the two values on the top of the stack is removed and replaced by $s_1 - s_0$.

1.3 The library

The library has a class `CodeFile` that is the base class for creating a program with "runnable" sequences of instructions, i.e. a (binary or `.bin`) file, executable by the virtual machine or more specifically by the virtual machine interpreter.

To create such a bytecode file, one must instantiate that class `CodeFile` and **add** procedures, structs and global variables using the procedures of the `CodeFile` object. Objects that represent local variables and instructions are created using the library and added to the procedure objects, which are of class `CodeProcedure`. When the program is complete, that is, when all the elements of the program has been added to the `CodeFile` object and its `CodeProcedure` objects, the actual bytecode can be extracted by the procedure `getBytecode()` of a `CodeFile` object. The array of bytes that is then created, is usually written to a file, which can then be read by the virtual machine and run by its interpreter.

A typical use of an instance of `CodeFile` will be something like this:

```
CodeFile codeFile = new CodeFile();
String filename = "example.bin";

// Building the bytecode with instructions
// like the following:
codeFile.addProcedure("Main");
CodeProcedure main =
   new CodeProcedure("Main",
                     VoidType.TYPE,
                     codeFile);
main.addInstruction (new RETURN());
```

```
codeFile.updateProcedure(main);

// ... and more ....

byte [] bytecode = codeFile.getBytecode();   // bytecode = array of bytes.
DataOutputStream stream =
   new DataOutputStream(new FileOuputStream(filename));
stream.write(bytecode);
stream.close();
```

Listing 1: Typical use of the `Codefile` class

In that example, first an object of the class `CodeFile` is created. Then, the procedure `"main"` is added. More procedures, structs, global variables, and constants may be added to it. Then one can get the bytecode, which is an array of bytes, and write it to a file, as shown in the code snippet.

The generated bytecode file can then be inspected, for instance with an editor for binary files like the Eclipse Hex Editor Plugin (EHEP).[2]

1.4 Using the virtual machine

A bytecode file can be used in two ways, file, both offered by the class `VirtualMachine` of the `runtime` package. Either the execution is triggered *externally*, from the **command line**. Or internally from within a Java program. For the exernal use, the command line commands are

```
java runtime.VirtualMachine <FILENAME>
```

Using the interpreter from inside a program, one needs to instantiate the class `VirtualMachine` and call its `run` method.

```
VirtualMachine vm = new VirtualMachine("<FILENAME>");
vm.run();
```

The class `VirtualMachine` can also list the content of the bytecode file in textual form. Eexternally with the command line option `-l` or, from inside a program, using the `list` method:

```
java runtime.VirtualMachine -l <FILENAME>
```

resp.

```
VirtualMachine vm = new VirtualMachine("<FILENAME>");
vm.list();
```

As an example, assume a program like this

```
// File: ./Simple.d
struct Complex {
  var float Real;
  var float Imag;
}

var Complex dummy;
func Main () { }
```

[2] For Ehep, see `http://ehep.sourceforge.net/` or `https://marketplace.eclipse.org/content/ehep-eclipse-hex-editor-plu`

Listing the generated bytecode with the -l option would then look as follows.

```
Loading from file: ./Simple.bin
Variables:
0: var Complex dummy
Procedures:
0: Main()
   0: return
Structs:
0: Complex
     0: float
     1: float
Constants:
STARTWITH: Main
```

1.5 Source code overview

The bytecode library is a package contain a few classes and besides that sub-packages with their own classes. The most important classes at the top-level the library are the following

class	purpose
`CodeFile`	main class to create byte code
`CodeProcedure`	create code for procedures
`CodeStruct`	create code for structs

The first one is the the main class for creating byte code, mentioned earlier, the other two for creating a procedure resp. a struct in the byte code.

There are two sub-packages, contained in corresponding sub-directories

sub-package	purpose
`instructions`	byte code instruction set
`type`	byte code representation of the types

The packages are referred to accordingly `bytecode.instructions.*` and `bytecode.type.*`. The instructions in the first package are represented as concrete subclasses of the abstract `Instruction` class. Similarly for the types (as subclasses of the abstract `CodeType` class. That package contains all the types used in the byte code.

Although usually not directly used by a programmer, it might be instructive to know the main classes of the virtual machine resp. the run-time system, as well. See the package resp. directory `runtime` The main classes are the following:

class (in `runtime`)
`VirtualMachine`
`Interpreter`
`Stack`
`Heap`
`ActivationBlock`
`Loader`

The virtual machine class is the starting point for running a program, as mentioned earlier. The loader class loads a a program from a file. The actual interpretation of the byte code is provided by the interpreter class. The classes `Heap` and `Stack` are responsible for management of the corresponding memory. Note that there is a single stack for each program (as the programs are single-threaded). The heap contains structs, and their allocation and access is maintained by the mentioned class. The class for activation blocks handles and stores the program counter, local variables etc. and also handles the call and return of a procedure in conjuction with the interpreter.

[Add some more from the original]

2 Building a complete program

We have shown the basics of how to build a bytecode program (binary file) using the bytecode library. In this section we will show some of the details by covering each of the classes of the library. Details about all the instructions will come in Section 5.

The main parts for creating a new program are: create an object of the class `CodeFile`. Afterwards add the procedures, i.e., objects of class `CodeProcedure`, structs, i.e., objects of class `CodeStruct`, and so on. Finally, when calling the `getBytecode()` procedure, the bytecode library will generae the actual bytes from the object hat have been created and from the "properties" given to those objects. Note that there a *four* steps to create a procedure (or a struct, or global variable).

1.	*add* the definition to the `CodeFile` object	`addProcedure`.
2.	*create* the `CodeProcedure` object (say `p`)	`new CodeProcedure`.
3.	*add* "properties", such as instructions, to `p`	
4.	*update* the `CodeProcedure` object in the `CodeFile`	`updateProcedure`

These four standard steps are illustrated in slightly more detail in the following example.

2.1 Creating code for a procedure: A small example.

The example code creates a "code file" and first adds the name of a *library* procedure to be used. It then adds 4 items, in preparation for the follow-up steps, which add the code for those items. Added is procedure `Main`, a global variable `myGlobalVar`, a procedure `test` and a struct `Complex` (I).

In the next steps, the procedure `Main` is properly defined and the information is added. The main procedure has the void return type, no parameters, no local variables and the body consists of a single instruction, namely `return`. The code of part (II) corresponds to the steps 2.-4. for the 4 steps mentioned above, the first step for the procedure has been done in "section" (I).

The global variable is typed with the struct type `Complex` (III).

The procedure `test` has 2 parameters (of type float and a reference type to `Complex`). The procedure loads the first parameter onto the stack and then calls the (library) procedure `print_float` to print its value (IV). The struct `Complex` is created and the two fields, both of floating point type, are added to it (V). The procedure `print_float` used in the program is a library procedure. Nonetheless, it needs to be added, but without instructions (VI). See also Section 4.4. Finally, the mandatory main method is set, before the bytecode can be extracted to a file (VII). Section 2.2 later shows the generated machine code for this example.

```
// Make code:
CodeFile codeFile = new CodeFile();
codeFile.addProcedure("printFloat")

//-------- (I) --------------------
codeFile.addProcedure("Main");
codeFile.addVariable("myGlobalVar");
codeFile.addProcedure("test");
codeFile.addStruct("Complex");
//-------- (II) ----------
CodeProcedure main =
    new CodeProcedure("Main",
                      VoidType.TYPE,
                      codeFile);
main.addInstruction(new RETURN());
codeFile.updateProcedure(main);

//-------- (III) ----------
codeFile.updateVariable("myGlobalVar",
                        new RefType(codeFile.structNumber("Complex")));
//-------- (IV) ----------
CodeProcedure test = new CodeProcedure("test",
                                       VoidType.TYPE,
                                       codefile);
test.addParameter("firstPar", FloatType.TYPE);
test.addParameter("secondPar", new RefType(test.structNumber("Complex")));
test.addInstruction(new LOADLOCAL(test.variableNumber("firstPar")));
test.addInstruction(new CALL(test.procedureNumber("print_float")));
test.addInstruction(new RETURN());
codeFile.updateProcedure(test);

//-------- (V) ----------

CodeStruct complex = new Codestruct("Complex");
complex.addVariable("Real", FloatType.TYPE);
complex.addVariable("Imag", FloatType.TYPE);
codeFile.updateStruct(complex);

//-------- (VI) ----------
CodeProcedure printFloat = new CodeProcedure("print_float",
                                             VoidType.TYPE,
                                             codeFile);
test.addParameter("f", FloatType.TYPE);
codeFile.updateProcedure(printFloat);

//-------- (VII) ----------
codeFile.setMain("Main");
byte[] bytecode = codeFile.getBytecode();
//.............. write the bytes to a file
```

Listing 2: 4 steps to generate code for a procedure

2.1.1 Class `CodeFile`

This is the class byecode is created from and all elements must be added to the corresponding object. It also provides the service of returning indices to the given elements, as we will see later. Those indices are needed when instruction classes are created. They reference the elements within the program. Adding something to a `CodeFile` object is done in *two* stages; first, the element is added using something like the `addProcedure` procedure, supplying only the name. Then later, the `updateProcedure` must be called with a reference to the actual procedure object. After a procedure has been added (but before it has been updated) its index can be found and used, for example to create a call to it, as we will see.

An object of the `CodeFile` class is typically seen by all the nodes in the abstract syntax tree, by for example passign around a reference to it. An element in the syntax tree is typically responsible for adding itself to the compiled result by using the procedures of the `CodeFile` or `CodeProcedure` classes.

A *global variable* is added by using the procedure `void addVariable(String name)`. After a global variable has been added, its index (`id`) in the program may be found using its name, calling the procedure `int globalVariableNumber(String name)` The type of the variable *must* be supplied *before* the bytecode is generated. The generation is done by calling the method `void updateVariable(String name, CodeType type)`. All global variables must have unique names.

A *procedure* is added by using the procedure `void addProcedure(String name)` After a procedure has been added, its index (`id`) in the program can be found using its name, calling the method `int procedureNumber(String name)`. The details of the procedure must be supplied before the bytecode is generated. That is done by invoking the method `public void updateProcedure(CodeProcedure codeProcedure)`.

For a struct, there are analogous procedures `public void addStruct(String name)`, `int structNumber(String name)`, and `void updateStruct (CodeStruct codeStruct)`. In addition, getting the index of a field in a struct is done by calling `int fieldNumber(String structName, String varName)` using the name of the struct and the name of the field.

A `string constant` is added by using the procedure `int addStringConstant(String value)`. Note that this procdure returns the index (`id`) of the constant *directly* and there is no procedire to fetch the index of a constant later. The index is used for string literals by the compiler.

After all the elements are added, it is important to let the interpreter know which procedure to start with. This is done by using the the name of the procedure (typically `"main"`) and calling `void SetMain(String name)`.

2.1.2 Class `CodeProcedure`

A procedure in the program is made by first adding its name to the `CodeFile` object, then creating an object of this class, then adding the parameters, local variables, and instructions to the object, and finally by updating the `CodeFile` object with the `CodeProcedure` object.

A *procedure object* is creatd by the constructor `CodeProcedure(String name, CodeType returnType, CodeFile codeFile)`. This takes the unique name of the procedure, the return type (see `CodeType` below) and the code file that it will be added to. The reason why the code file is included is that it is needed by the procedure object to provide some of the code file services (relying on a delegation pattern mechanism).

Parameters and *local variables* are added by the procedures `void addParameter(String name, CodeType type)` and `void addLocalVariable(String name, CodeType type)`.

An *instruction* is added to the procedure object with `int addInstruction(Instruction instruction)`. The return value of this method is the *index* of the instruction in the procedure's list of instructions. Sometimes one wants to *replace* and earlier instruction. This is done by using `void replaceInstruction(int place, Instruction instruction)`. For instance, one may insert temporarily a `NOP` instruction to be replaced later by a jump instruction `JMP`. Read more in Section 4.2 on jumps. A procedure body cannot be completely empty, it must at least contain one final `RETURN` instruction.

The *index* of a variable or a parameter can be found by using `int variableNumber(String name)`. Note that local variables mus have unique names in any block, and that includes the parameters. The parameters are given the first indices from left to right, starting from `0`. The (remaining) local variables are given the subsequent indices in the order of their declaration.

A `CodeProcedure` object can find the indices of elements using its `CodeFile` object, so it also has the procedures `globalVariableNumber`, `procedureNumber`, `structNumber`, and `fieldNumber`. It also has and delegates the `addStringConstant` procedure.

2.1.3 Class `CodeStruct`

A *struct* is created with the constructor `CodeStruct(String name)` providing the name of the struct. A field is added to the struct by using `void addVariable(String name, CodeType type)`. To retrieve the index of a field added to the struct, one may use `int fieldNumber(String name)`. See also the `fieldNumber` procedure of `CodeFile`.

2.1.4 Subpackage for types

Class `CodeType` This is an abstract class and it has as concrete subclasses the different classes of types: `VoidType`, `BoolType`, `IntType`, `FloatType`, `StringType` and `RefType`. The void type is used for procedures that don't return a value and the reference type is for references to structs (i.e., records).

The basic types have a *singleton* object (for instance `StringType.TYPE`), which is used as actual parameter whenever needed, for example to define the return type of a procedure or the type of a field in a struct.

The `RefType` class is a little different. There is no singleton and its constructor has an integer parameter which is the *index* of the struct for which this type is a reference. The `RefType` is use by creating an object with the index (`id`) for the struct as the single parameter. One may make many such objects for the same type (the same index) if that is more convenient, or just reuse the same object for the type. An object of the reference type to the struct `"Complex"` mentioned before, for example, can be created like this.

```
CodeFile cf = <...> ;
...
cf.addStruct("Complex");
...
RefType rt = new RefType(cf.structNumber("Complex");
```

2.2 Virtual machine listing

The following shows the code corresponding to the example from earlier in the section.

```
Loading from file: ./example.bin
Variables:
```

```
0: var Complex myGlobalVar
Procedures:
0: func void print_float()
1: func void Main()
   0: return
2: func void test(float 0, Complex 1, float 2)
   0: loadlocal 0
   3: call printfloat {0}
   6: return
Structs:
0: Complex
   0: float
   1: float
Constants:
STARTWITH: Main
```

3 How the interpeter works

When the virtual machine is started, the interpreter is set up by the *loader.* It has a variable pool, which holds the type of each global variable. It has a procedure pool, which contains all the procedures: their parameter and local variable types, return type, and instructions. It has a struct pool with the layout of the structs; their names and the types (but not names) of the fields. It also has a constant pool with all the constants from the byte code file. All these pools are indexed by numbers (`id` s) which are the numbers used by the instructions.

When the interpreter is started, space is allocated for the global variables and they are initialized with the initial values for their types (see further down for more information about initial values). The a *stack* and a *heap* are created and an *activation block* for the main procedure is created as well. Then the interpreter starts interpreting the byte code of that procedure at the first byte, setting the program counter `pc` to 0

The instructions do things like loading a global variable onto the stack (`LOADGLOBAL`), The `LOADGLOBAL` instructions has 2 extra bytes which contain the id of the variable to push to the stack from the global variables. When that instruction is performed, the program counter `pc` must be incremented by 3 to move to the next instruction. The increment differs from instruction to instruction and is listed as the *size* in the table with all instructions in Section 6. Another instruction is `ADD` for addition. When that is interpreted, the two values on top of the stack are added up. Which kind of addition is done depends on the types of the two summands on the stack, determined at run-time. The result of the addition is pushed to the stack, and since the size is only one (the instruction byte only), 1 is added to the program counter.

Block levels are not supported by the virtual machine and only either global or else local variables can be accessed (`LOADGLOBAL`, `LOADLOCAL`, `STOREGLOBAL`, `STORELOCAL`). All names for procedures, structs, and variables must be *unique.* A procedure must always end with a `RETURN` instruction. If a procedure found in the binary file at loadtime is without instructions, it is assumed to be a library procedure and a cll to it results in a lookup using a table of library procedures.

All variables (global and local ones, as well as field in structs) are allocated with initial values, which depend on their types. An integer is set to 0 a float to 0.0, a string to the empty string, and a reference is set to the null reference.

3.1 Calling a procedure

A procedure is called with the `CALL` insturction. The byte instruction is followed by the index of the procedure being called. The interpreter locates the procedure by using the intext, creates an *activation block* from the information it has, initializes the local variables, saves the program counter, and sets the program counter to the first byte of the called function.

3.2 Return

The activation blcok is popped off the stack and the program counter is set to where it was before the corresponding call. The return value, which the called precedure left on the statck, is again left on the top of the stack for the calling procedure.

3.3 Jumping

Jumping is simply done by setting the program counter to the byte with the number that accompagnies the jump instruction. This is always a local address *within* a procedure's instruction bytes.

3.4 Treating records

3.4.1 Allocating a struct on the heap

When a struct is allocated on the heap, using the `NEW` instruction, a reference is left on the stack that can be passsed around and saved in variables. The `NEW` instruction is followed by the index of the struct type to allocate.

3.4.2 Get and put fields

The instruction `GETFIELD` is followed by the index of the struct and the index of the field within the that stuct. When it is interpreted, the interpreter assumes that a reference to the struct is on top of the stack and that reference is then popped off from the stack, The heap is instructed to get the value of the field witin the struct and the interpreter pushes the value of the field onto the stack, If the reference is the null reference, the interpreter aborts with an error message.

4 Some typical tasks

This section show some of the usual tasks. As already shown, one can obtain instructions by instantiating the corresponding class and supplying one or two integer values which are the ids of procedures, structs, variables or something to be used when the intruction is interpreted. For instance, the `JMP` instruction is created with an integer parameter which is the index of the instruction to be jumped to. When an instruction is added to the stream of bytes, it is followed by these indices coded into 0 to 4 bytes, depending on the size needed. In this way a "byte" can be 1 to 5 bytes long.

4.1 Calling a procedure

The constructor of the `CALL` class has an integer parameter `funcNum`. That is the index of the procedure and can be gotten from a `CodeFile` object if the procedure has been added. So a call instruction is created and added to the list of instructions. See the corresponding line in the example of Listing 2:

```
test.addInstruction(new CALL(test.procedureNumber("print_float")));
```

4.2 Jumping

The constructore of the `JMP` class has an integer parameter `jumpTo`. This is the index that the instraction has in the list of the instructions of this procedure. To way to get the index of an instruction is to save the integer returned from the `addInstruction` method. A trick fro placing *labels* in the code is to add a dummy instruction `NOP` at a place where one wants to insert a jump or wants to jump to. For example, the following creates the code for an infinite loop.

```
int top = test.addInstruction(new NOP());
// here code for the body of the infinite loop could be added.
test.addInstruction(new JMP(top));
```

Listing 3: Infinite loop

Important: the numbers used in the cosntructor for `JMP` and the conditional jump classes are the index of the instruction in the list of instructions. In this list, all instructions are considered to have size one. This is so that there will be no problems when replacing an instruction with another of a different size. When the bytecode is created and a new number is calculated and replaces that number (for all jumps) with the actual address withing the byte array, since at run-time the instructions with accompagnying operand values have different sizes.

4.3 Conditional jumps

The work simular to unconditional jumps, but there has to be a boolean value on the stack, when executed. Whether or not the jump is executed depends on the value of that boolean. For instance, the following creates the code for a do-while loop. More concretely for a loop of the form in some pseudo-code:.

```
do {
// loop body
} while (i<2);
```

```
int start = test.addInstruction(new NOP());   // jump target = loop start
                                              // statements for loop body....
test.addInstruction(new LOADLOCAL(test.variableNumber("i")));
test.addInstruction(new PUSHINT(new Integer(2)));
```

```
test.addInstruction(new LT());              // boolean value now on the stack
test.addInstruction(new JMPTRUE(start));    // jump back if true
```

Listing 4: do-while loop

4.4 Library procedures

When library procedures are needed, they may be added to the `CodeFile` (and updated, the name alone is not enough), but no instruction should be added, The interpreter recognizes the use of a library procedure by the fact that it contains no instruction in the binary file (`CodeFile`).

5 Finally, remember this.

To sum up, here are some important points to remember

- Always add a return statement to the end of the instructions of a procedure.
- Always set the main method.
- add the library procedures (like `print_int` etc.) as procedures, but without instructions
- if the reference on the stack is a null reference when trying to access a field of it, the interpreter will print the error `Nullpointer at GETFIELD` or the equivalent message for `PUTFIELD`, and the virtual machine will abort.
- do not hust add, but remember to update procedures, structs, and global variables.
- use the *list* option (`-l`) to inspect your bytecode and even take a look at it with an hex editor.

6 Instructions

Below is the table with all instructions supported by the virtual machine and that can be found int the bytecode library. We use s_0 for the top of the stac, s_i for the next element, and so on. When the symbol † (dagger) is found after the tanme of the instruction, we mean that are are more details on the types of what is on the stack at the end of this section (look up the instruction there).

6.1 Summary of the instructions

6.1.1 Binary operators

They require two values on the stack and leave one there. They have *no* extra value. There are the following 14 binary operators: `ADD`, `AND`, `DIV`, `EQ`, `EXP`, `GT`, `GTEQ`, `LT LTEQ`, `MUL`, `NEG`, `NOR`, `OR`, `SUB`.

6.1.2 Unary operators

The numbers correspond to the *opcodes*.

operator	nr.	extra bytes	before	after
`ADD` †	01	none	first operand s_1, second operand s_0, both: int, float, or string	$s_1 + s_0$
`AND`	02	none	first operand s_1, second operand s_0, both: bool	$s_1 \wedge s_0$
`CALL` †	03	2 bytes (short), with the index (id) of the function	the parameters from the left s_n to the right s_0	value returned, if any
`DIV` †	35	none	the dividend s_1 and the divisor s_0, both int or float	s_1/s_0
`EQ` †	04	none	first operand s_1, second operand s_0, both: int, float, or bool	a boolean
`EXP` †	05	none	first operand s_1, second operand s_0, both: int or float	a float, result of $s_1^{s_0}$
`GETFIELD`	06	4 bytes (2 shorts), index of the field within the struct and the index (id) of the struct	reference to the struct s_0	the value of the field, if s_0 is not a null reference
`GT`	07	none	first operand s_1, second operand s_0. Both: int or float	a boolean, result iff $s_1 > s_0$
`GTEQ`	31	none	first operand s_1, second operand s_0. Both: int or float	a boolean, result iff $s_1 \geq s_0$
`JMP`	08	2 bytes (short) with the position in the bytes of the function to jump to	none	none
`JMPFALSE`	09	2 bytes (short) with the position in the bytes of the function to jump to	a boolean s_0. Jump only if false.	none

Continued on next page

Continued from previous page

operator	nr.	extra bytes	before	after
`JMPTRUE`	10	2 bytes (short) with the position in the bytes of the function to jump to	a boolean s_0. Jump only if true.	none
`LOADGLOBAL`	11	2 bytes (a short) with the index (id) of the global variable to load.	none	the value of the global variable.
`LOADLOCAL`	12	2 bytes (a short) with the index (id) of the local variable to load. Remember params!	none	the value of the local variable.
`LOADOUTER`	13	4 bytes	**Not implemented in this version. No support for block structure!**	
`LT`	29	none	first operand s_1 and second operand s_0. Both: int or float	a boolean: true iff $s_1 < s_0$.
`LTEQ`	30	none	first operand s_1 and second operand s_0. Both: int or float	a boolean: true iff $s_1 \leq s_0$.
`MUL` †	34	none	first operand s_1, second operand s_0, both: int, float	$s_1 * s_0$
`NEQ` †	32	none	first operand s_1, second operand s_0, both: int, float, or bool	a boolean, result of $s_1 \neq s_0$
`NEW`	14	2 bytes (a short) with the index (id) of the struct to create an instance of.	none	a reference to the newly created struct.
`NOP`	15	none	none	none, the instruction does nothing
`NOT`	16	none	a boolean, s_0	a boolean, $\neg s_0$

Continued on next page

Continued from previous page

operator	nr.	extra bytes	before	after
`OR`	02	none	first operand s_1, second operand s_0, both: bool	$s_1 \vee s_0$
`POP`	28	none	some value s_0	none, instruction removes the top
`PUSHBOOL`	18	1 byte with the constant value 1 (true) or 0 (false)	none	the boolean constant from the extra byte
`PUSHFLOAT`	19	4 bytes with the value of the float constant	none	the float constant from the extra bytes
`PUSHINT`	20	4 byte with the value of the float constant	none	the integer constant from the extra bytes
`PUSHNULL`	21	none	none	a null reference
`PUSHSTRING`	22	2 bytes (a short) with the index (id) of the string constant	none	the string constant
`PUTFIELD`	23	4 bytes (2 shorts) which are the index of the field within the struct and the index (id) of the struct.	the vaue to assign to the field s_1 and the reference to the struct s_0	none
`RETURN`	24	none	a return value s_0 if the procedure has one	**not applicable**
`STOREGLOBAL`	25	2 bytes (a short) with the index (id) of the global variable to store to	the value s_0 to store into the global variable.	none
`STORELOCAL`	26	2 bytes (a short) with the index (id) of the local variable to store to. Remember params!	the value s_0 to store into the local variable.	none
`STOREOUTER`	27	4 bytes	**Not implemented in this version. No support for block structure!**	

Continued on next page

Continued from previous page

operator	nr.	extra bytes	before	after
`SUB` †	33	none	first operand s_1, second operand s_0, both: int or float	$s_1 - s_0$