UNIVERSITY OF OSLO
Department of Informatics

Interpreter and bytecode for INF5110

Fredrik Sørensen,
Stein Krogdahl and
Birger
Møller-Pedersen

26th May 2008

Contents

1 Introduction **2**

1.1 The interpreter . . . 2

1.2 Interpreting bytecode . . . 3

1.3 The library . . . 3

1.4 Using the Virtual Machine . . . 5

1.5 Source code overview . . . 6

2 Building a complete program **7**

3 How the interpreter works **13**

4 Some typical tasks **15**

5 The instructions **19**

6 Finally – Remember this **27**

1 Introduction

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

Bytecodes in general are similar to machine code, but instead of being run directly on a machine, they are usually interpreted by a bytecode interpreter. They may also be translated into machine code before being run. A bytecode construction library is a library used to simplify the task of generating bytecode for a bytecode interpreter. Bytecode is named exactly that because each instruction is the size of a byte.

Our bytecode is very similar to Java bytecode and our bytcode library is based on the Byte Code Engineering Library (BCEL)[1]. We chose to write a simpler, stripped down version that does not support classes, virtual procedures and all the things Java's bytecode has. We did this so that it would be simpler to work with and there would be less code. For example, one does not need to create classes like, since our language does not have them, just for "some unexplained reason". Still, we encourage the students to look at BCEL, because it is a nice tool and very well designed.

The bytecode library and the interpreter are both written in Java. All the code is handed out in source form and consists of the packages:

- **bytecode.***: the classes to create bytecode
- **bytecode.instructions.***: instruction classes (supporting the above)
- **bytecode.type.***: type classes (supporting the above)
- **runime.***: all the classes of the runtime system

1.1 The interpreter

The interpreter is stack based and it interprets the about 40 different instructions of our bytecode. The interpreter "automatically" handles allocation of

[1] http://jakarta.apache.org/bcel/

struct types, method calls and access of variables in a struct when instructed to, by the bytecode instructions. In the first version it does not have a garbage collector and allocates memory sequencially as long as there is memory left for it.

The operators (instructions) are so-called “stack operations”. It means that when the interpreter executes an instruction it pops a number of operands (zero or more) from the stack, performs the task required by the instruction and leaves a number (usually just one) of result values on the stack.

The interpreter is not written for efficiency, but rather for readability and ease of creating runnable bytecode. For example, the types are kept on the stack together with the calculated 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 (addition) instruction will be performed differently if there are two integers on the stack, two floats, or one of each. This is 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 converts an int to a float).

1.2 Interpreting bytecode

As mentioned, the interpreter is stack based. The parameters from right to left in an operation (including user defined functions) must be placed on the stack with the leftmost on the bottom and the rightmost on the top of the stack before the instruction itself is interpreted. For example, the *SUB* (subtraction) instruction requires that the two operands the *SUB* instruction will be applied to are on the stack and in the right order beforehand. The number to subtract must be on the top and the number to subtract from below it. 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 for creating a program with “runnable” sequences of instructions, that is, a (binary or .bin) file that can be executed by the virtual machine or more specifically by the virtual machines interpreter.

To create such a bytecode file, one must create an object of the 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 using 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 the CodeFile class will be something like this:

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

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

// ... and more ...

byte[] bytecode = codeFile.getBytecode();
DataOutputStream stream = new DataOutputStream(
                          new FileOutputStream (filename));
stream.write(bytecode);
stream.close();
```

In the example, an object of the class *CodeFile* is first created. Then, the procedure “main” is added to it. More procedures, structs, global variables and constanst may be added to it. Then one can get the bytecode (an array of bytes) and write it to a file, as shown.

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

[2]http://ehep.sourceforge.net/

1.4 Using the Virtual Machine

There are two ways to run a program from a bytecode file. Both are with the class *runtime.VirtualMachine*.

- One is to use the command line and write

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

- The other is to use the class VirtualMachine in a program and create an object of it and call the "run" procedure.

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

The class VirtualMachine can also list the content of the bytecode file in a textual form. This also, can be done in two ways:

- One is to use the command line and write

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

- The other is to use the class in a program and create an object and call the list procedure.

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

If we had a program like this ...

```
// File: ./Simple.d
struct Complex {
    var float Real;
    var float Imag;
}
var Complex dummy;
func Main() {  }
```

... listing the generated bytecode with the “-l” option would look like this:

```
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 most important classes and packages of the bytecode library are *CodeFile* (the main class for creating bytecode), *CodeProcedure* (the class for creating a procedure in the bytecode), *CodeStruct* (the class for creating a struct in the bytecode), *bytecode.instructions.** (the package with all the bytecode instruction classes) and *bytecode.type.** (the package with all the classes for the types used in the bytecode).

Although they are usually not used by a programmer, it is nice to know the main classes of the Virtual Machine. It could be useful to look at the Virtual Machine code, and one may need to add to it or debug it. The main classes of the runtime system (the Virtual Machine) are *VirtualMachine* (the starting point for running a program), *Loader* (the class that loads the program from a file), *Interpreter* (the class that actually does the interpretation of the bytecode), *Stack* (the class that handles the single stack in a program), *Heap*,(the class that allocates, stores and retrieves the structs and their fields) and *ActivationBlock* (the class that handles and stores the program counter, local variables and so on and handles the call and return of a procedure in conjunction with the Interpreter).

2 Building a complete program

We have showed the basics of how a bytecode program (binary file) is built using the bytecode library. In this section we will show some of the details by covering each of the classes in the library and what they can do. Details about all the instructions will come in section 5.

The main parts of making a new program are; making an object of the class *CodeFile*, then adding the procedures (objects of class *CodeProcedure*) and structs (objects of class *CodeStruct*) and so on. Finally, when the *getBytecode()* procedure is called, the bytecode library will generate the actual bytecode bytes from the objects that has been created and the properties given to those objects.

Note that there are four parts to making a procedure (or struct or global variable):

1. Adding the definition to the CodeFile object (*addProcedure*).
2. Creating the CodeProcedure object (*new CodeProcedure*).
3. Adding the properties, like the instructions, to the CodeProcedure object.
4. Updating the CodeProcedure object in the CodeFile (*updateProcedure*).

To see the details of these four steps, read the following example.

A small example

Below is a small example. All the classes and procedures used in this example will be explained later in this section. The example code creates a program and first adds the name of a library procedure to be used. It the adds a procedure "Main", a global variable "myGlobalVar", a procedure "test" and a struct "Complex" (I). The procedure "Main" has return type "Void", no parameters, no local variables and only has a single instruction RETURN (II).

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

The procedure "test" has two parameters; one of type Float and one of reference type Complex. The procedure loads the first parameter onto the stack and the calls the procedure "print_float" to print the value (IV).

The struct “Complex” is created and the two fields, both of type Float, are added to it (V).

The procedure print_float must be added, but without instructions. Read more about library procedures in section 4 (VI).

Finally, the main method must be set before the bytecode can be extracted and written to a file (VII).

At the end of the section is a listing of the bytecode file created from this.

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

// 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.
```

CodeFile

This is the class the bytecode is created from and all the elements of the program must be added to this object. It also provides the service of returning indices given to the elements, as we will see used later. These 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 passing 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 by using its name, calling the procedure `int globalVariableNumber (String name)`. The type of the variable *must* be supplied before the bytecode is generated. It is done by calling `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 may be found by using its name, calling the procedure `int procedureNumber( String name)`. The details of the procedure must be supplied before the bytecode is generated. It is done by calling `public void updateProcedure( CodeProcedure codeProcedure)`.

For a **struct** there are similar procedures `public void addStruct(String name)`, `int structNumber(String name)` and `void updateStruct(Code-Struct 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 addStringCon-stant(String value)`. Note that this procedure returns the index (id) of the constant directly and there is no procedure to fetch the index of a constnt later. The index is used when one wants to access a defined constant and push it on the stack. This is used for string literals by a 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 name of the procedure (typically "main") and calling `void setMain(String name)`.

CodeProcedure

A procedure in the program is made by first adding the 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 created by using 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 codefile that it will be added to. The reason that the codefile is included is that it is needed by the procedure object to provide some of the codefiles services through delegation.

A **parameter** or **local variable** is added by using the procedures `void addParameter(String name, CodeType type)` or `void addLocalVaria-ble(String name, CodeType type)`.

An **instruction** is added to the procedure object by using `int addInstruc-tion(Instruction instruction)`. The return value of this is the index of the instruction in the procedures list of instructions. Sometimes one wants to replace an earlier inserted instruction. This is done by using `void replace-`

`Instruction(int place, Instruction instruction)`. For example, ont may insert a NOP instruction, to later be replaced by a JMP. Read more in section 4 on jumps. A procedure must have at least one instruction, and also, that instruction has to be a final RETURN instruction.

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

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*.

CodeType

This is an abstract class and it has as concrete subclasses the different classes of types: *VoidType* (used when a procedure has no return type), *BoolType*, *IntType*, *FloatType*, *StringType* and *RefType* (When the type is a reference to a struct).

The basic types have a singleton object (for example `StringType.TYPE`) which is used as an actual parameter whenever that is 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 used 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 (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" can be created like this:

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

Virtual Machine listing of the example from earlier in this 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 print_float {0}
    6: return
Structs:
0: Complex
    0: float
    1: float
Constants:
STARTWITH: Main
```

3 How the interpreter 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 bytecode file. All these pools are indexed by numbers (ids), which are the numbers used in 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 on initial values). Then a stack and a heap is created and an activation block for the main procedure is created and the interpreter starts interpreting the bytecode of that procedure at the first byte (Setting the program counter or pc to 0).

The instructions do things like load a global variable (LOADGLOBAL) onto the stack. The LOADGLOBAL instruction 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, 3 must be added to the pc to move to the next instruction. This increment differs from instruction to instruction and is the "size" in the table with all the instructions in section 5. Another instruction is ADD. When that is interpreted, the two values on top of the stack is added to each other. What kind of addition depends on the types of the two, which is determined at runtime. The result is pushed on the stack and since the size is only one (the instruction byte only), one is added to the pc.

Block levels are not supported by the virtual machine and only global or local variables can be used (LOADGLOBAL, LOADLOCAL, STOREGLOBAL, STORELOCAL). All names of 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 call to it results in a lookup in a table of library procedures.

All variables; in structs, global and local are allocated with initial values; which depend on their types. An int is set to 0, a float to 0.0, a string to "" and a reference is set to the Null Reference.

Calling a procedure

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

Return

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

Jumping

Jumping is simply done by setting the program counter to the byte with the number that accompanies the jump instruction. This is always a local address within a procedures instruction bytes.

Allocating a struct on the heap

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

Get and Put Field

The intstruction GETFIELD is followed by the index of the struct and the index of the field within that struct. When it is interpreted, the interpreter assumes that a reference to the struct is on the top of the stack, and that reference is popped of the stack. The heap is instructed to get the value of the field within the struct and the interpreter pushes the value of the field to the stack. If the reference is the Null Reference, the interpreter is aborted with an error message.

4 Some typical tasks

In this section we show how some of the usual tasks are solved.

As has already been shown, some of the instruction classes are created by supplying one or two integer values, which are the ids of procedures, struct, variables, or something, that are to be used when the instruction is interpreted, for example, JMP is created with an integer parameter, which is the index of the instruction to jump to.

When an instruction is added to the stream of bytes, it is followed by these indices coded as 0 to 4 bytes, depending on the size needed. In this way, a "byte" can be from 1 to 5 bytes long.

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 like in the example in the previos section.

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

Jumping
The constructor of the JMP class has an integer parameter jumpTo. This is the index that the instruction has in the list of the instructions of this procedure. The way to get the index of an instruction is to save the integer returned from the addInstruction procedure. A trick for 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 code for an infinite loop.

```
int top = test.addInstruction(new NOP());

// The statements of the infinite loop

test.addInstruction(new JMP(top));
```

Important:
The numbers used in the constructors of 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 with replacing an instruction with another of a different size. When the bytecode is created a new number is calculated and replaces that number (for all jumps) with the actual address within the byte array, since at runtime the instructions (with accompanying values) have different sizes.

Conditional jumps

They are like jumps, but there must be a boolean value on the stack before it is interpreted. The jump is performed or not depending on the value of the boolean. For example, the following creates code for a do while loop.

```
// do {
//
// The statements inside the loop ...
//
// } while(i<2)

// Start of the loop
int start = test.addInstruction(new NOP());

// The instructions inside the loop ...

// First calculate the boolean expression.
test.addInstruction(new LOADLOCAL(test.variableNumber("i")));
test.addInstruction(new PUSHINT(new Integer(2)));
test.addInstruction(new LT());
// TRUE or FALSE is left on the stack.

// Jump back if true.
test.addInstruction(new JMPTRUE(start));
```

Sometimes a jump forward may need to be inserted. In that case a NOP is inserted and the index saved. Later it can be replaced by a jump or conditional jump instruction using `replaceInstruction( int place, Instruction instruction)`

Constants

Constants or literals are placed on the stack using the PUSH-methods, like PUSHINT(Integer). In the bytecode it is followed by 4 bytes (The integer constant) which is pushed on the stack. To push a string literal onto the stack, it first has to be registered as a constant and then its index is used with the PUSHSTRING instruction like below.

```
int constId = test.addStringConstant("Literal");
test.addInstruction(new PUSHSTRING(constId));
```

A float literal with value 0.0 is created like this:

```
test.addInstruction(new PUSHFLOAT(new Float(0.0)));
```

Working with structs

Instances of the structs are created with the NEW(Integer structNum) instruction. The parameter is the index (id) of the struct. A reference to the heap allocated struct is left on the stack by the interpreter. To assign a value to a field of a struct, the reference to the struct must be on the top of the stack and the value to assign to the field must be on the stack below that. When the PUTFIELD(int fieldNumber, int structNum) instruction is interpreted, the interpreter will locate the struct instance in the heap using the reference and set the field to the value found on the stack. Below is an example that creates a struct, saves it in a local variable named *cmplx* and sets one of its fields (*real*) to a float value (1.0).

```
test.addInstruction(new NEW(test.structNumber("Complex")));
test.addInstruction(new STORELOCAL(test.variableNumber("cmplx")));
test.addInstruction(new PUSHFLOAT(new Float(1.0)));
test.addInstruction(new LOADLOCAL(test.variableNumber("cmplx")));
test.addInstruction(new PUTFIELD(test.fieldNumber("Complex",
                                                  "real"),
                                  test.structNumber("Complex")));
```

Parameters and Local Variables

When a procedure is called. the parameters and local variables are all placed sequencially on the stack, starting with the leftmost of the parameters. All the parameters and local variables are accessed with LOADLOCAL(Integer), which places the value of the variable with the given index on the stack. The index starts with the leftmost parameter at 0 (the lowest on the stack) and ends with the last defined variable at N (the highest on the stack). **Note that usually we talk about 0 being the top of the stack**.

Here is a piece of code and how to generate the bytecode for line 3 only.

```
//1: func ret int add( int a, int b ) {
//2:   var int res;
//3:   res := a + b; // only bytecode for this line
//4:   return res;
//5: }

test.addInstruction(new LAODLOCAL(test.variableNumber("a")));
// test.variableNumber("a") returns 0

test.addInstruction(new LAODLOCAL(test.variableNumber("b")));
// test.variableNumber("b") returns 1

test.addInstruction(new ADD());

test.addInstruction(new STORELOCAL(test.variableNumber("res")));
// test.variableNumber("res") returns 2, the highest value
```

Library Procedures

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

5 The instructions

Below is a table with all the instructions that is supported by the virtual machine and that can be found in the bytecode library. We use s_0 for the top of the stack, s_1 for the next element and so on. When the symbol † (dagger) is found after the name of an instruction it means that there are more details on the types of what is on the stack at the end of this section (Look up the instruction there).

Summary of the instructions:

- **Binary operators**:
  They require two values on the stack and leave one there. They have no extra value. They are: **ADD, AND, DIV, EQ, EXP, GT, GTEQ, LT, LTEQ, MUL, NEQ, NOR, OR, SUB**

- **Unary operator**:
  It requires one value on the stack and leaves one there. It has no extra value. It is: **NOT**

- **Jump instructions**:
  They are followed by the address to jump to (a short). The conditionals require a boolean value on the stack. They are: **JMP, JMP-FALSE, JMPTRUE**

- **Procedure call instructions**:
  The call is followed by the id of the procedure (a short). Before a call, all the parameters must be on the stack with the leftmost at the bottom. Before a return, the return value (if any) must be on the stack. After a return from a call, the result value is on the top of the stack. They are: **CALL, RETURN**

- **Struct access**:
  They are followed by the id of the struct (a short) and the index of the field within it (a short). When using PUTFIELD, the value to be stored and the reference to the struct must be on the stack. When using GETFIELD, the reference to the struct must be on the stack and the value found in the field is left on the stack. They are: **GETFIELD, PUTFIELD**

- **Constants or literals**:
  They are followed by the value to be put on the stack (1-4 bytes).

In the case of PUSHSTRING it is the id of the string constant and in the case of PUSHNULL it is nothing. They are: **PUSHBOOL, PUSHFLOAT, PUSHINT, PUSHNULL, PUSHSTRING**

- **Local variables and parameters**:
  They are followed by the index (index) of the variable to fetch or store to. They are: **LOADGLOBAL, LOADLOCAL, STOREGLOBAL, STORELOCAL**
- **Struct allocation**:
  It is followed by the index of the struct to allocate and initialize. It is: **NEW**
- **The do-nothings**:
  They are followed by no extra value. NOP really does nothing. POP pops the top off the stack and discards the value. They are: **NOP, POP**

The operands for the current instruction on the stack are always popped off as part of interpreting the instruction. Unless otherwise mentioned, the result is always left on the stack.

Below is an alphabetic list of all the instructions, with the name of the instruction, the number (bytecode), any extra bytes needed, the operands that must be on the stack before execution and the result left on the stack.

Name	**Number**	**Extra bytes**	**On the stack before**	**Result on the stack**
ADD†	01	NONE	First operand (s_1) and second operand (s_0). Both: int, float or string.	Result of $s_1 + s_0$
AND	02	NONE	First operand (s_1) and second operand (s_0). Both boolean.	Result of $s_1 \& s_0$

CALL†	03	2 bytes (short) with the index (id) of the function.	The parameters from left (s_N) to right (s_0).	Value returned from procedure if any.
DIV†	35	NONE	Dividend (s_1) and divisor (s_0). Both int or float.	Result of s_1/s_0
EQ†	04	NONE	First operand (s_1) and second operand (s_0). Both: int, float or bool.	A boolean. True if $s_1 = s_0$, else false.
EXP†	05	NONE	First operand (s_1) and second operand (s_0). Both int or float.	A float, result of $s_1^{s_0}$.
GETFIELD	06	4 bytes (2 shorts) which are the 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) and second operand (s_0). Both int or float.	A boolean. True if $s_1 > s_0$, else false.

GTEQ	31	NONE	First operand (s_1) and second operand (s_0). Both int or float.	A boolean. True if $s_1 \geq s_0$, else false.
JMP	08	2 bytes (short) with the position in the bytes of this function to jump to.	NONE	NONE
JMPFALSE	09	2 bytes (short) with the position in the bytes of this function to jump to.	A Boolean (s_0). Jumps only if it is false.	NONE
JMPTRUE	10	2 bytes (short) with the position in the bytes of this function to jump to.	A Boolean (s_0). Jumps only if it is 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!**	N/A
LT	29	NONE	First operand (s_1) and second operand (s_0). Both int or float.	A boolean. True if $s_1 < s_0$, else false.
LTEQ	30	NONE	First operand (s_1) and second operand (s_0). Both int or float.	A boolean. True if $s_1 \leq s_0$, else false.
MUL†	34	NONE	First operand (s_1) and second operand (s_0). Both: int or float.	Result of $s_1 * s_0$
NEQ†	32	NONE	First operand (s_1) and second operand (s_0). Both: int, float or bool.	A boolean. True if $s_1 \neq s_0$, else false.
NEW	14	2 bytes (a short) with the index (id) of the struct to create and instance of.	NONE	A reference to the newly created struct.
NOP	15	NONE	NONE	NONE (It does nothing!)

NOT	16	NONE	A boolean (s_0).	A boolean. True if s_0 is false, else true.
OR	17	NONE	First operand (s_1) and second operand (s_0). Both boolean.	Result of $s_1 \mid s_0$
POP	28	NONE	Some value (s_0).	NONE. It just 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 constant from the extra bytes.
PUSHINT	20	4 bytes with the value of the integer constant.	NONE	The 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 value 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 in 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 in the local variable.	NONE
STOREOUTER	27	4 bytes.	**Not implemented in this version. No support for block structure!**	N/A
SUB†	33	NONE	First operand (s_1) and second operand (s_0). Both: int or float.	Result of $s_1 - s_0$

More details on some instructions Here or some comments for the instructions with a dagger (†) by the name.

ADD
The types of the two arguments can be any of int, float of string. If at least one of them is string, the result will be a string; the string concatenation of the two values. If none are string and at least one is float, the result will be a float; the sum of the two values. If both are int, the result will be an int; the sum of the two values.

CALL
If a formal parameter is of type float and the corresponding actual parameter on the stack is an int, it will be translated into a float.

DIV
If at least one of the operands is a float, the result will be a float; the floating point division of the two values. If both are int, the result will be an int; the integer division of the two values.

EQ
If one of the operands is a float and the other an int, the int value will be translated into a float before the two values are compared.

EXP
The operands can be int or float. The result will always be a float.

MUL
If at least one of the values is a float, the result will be a float; the multiplication of the two values. If both are int, the result will be an int; the multiplication of the two values.

PUTFIELD
If the type of the field is float and the actual value on the stack is an int, the value will be translated into a float.

RETURN
If the return type of the procedure is float and the actual result placed on the stack by the procedure is an int, the result will be translated into a float.

STOREGLOBAL
If the type of the variable is float and the actual value on the stack is an int, the value will be translated into a float.

STORELOCAL
If the type of the variable is float and the actual value on the stack is an int, the value will be translated into a float.

SUB
If at least one of the values is a float, the result will be a float; the subtraction of the second from the first. If both are int, the result will be an int; the subtraction of the second from the first.

6 Finally – Remember this

To sum up, here are some of the 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 (print_int, etc) as procedure, but without instructions
- If the reference on the stack is a Null Reference when one tries to access a field of it, the interpreter will print the error "Nullpointer at GETFIELD" or the equivalent for PUTFIELD, and the virtual machine will abort.
- Do not just add, but remember to update procedures, structs and global variables.
- Use this list option to see your bytecode and even take a look at it with a Hex Editor.