Assignment 7

CSCI 3136: Principles of Programming Languages

Due Apr 9, 2018

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

In this assignment, you will augment your parser of the TOY language so it translates any given TOY program into bytecode executable on a virtual machine I will provide to you next week. (The definition of the virtual machine's behaviour is given below.) I will also provide you with a "disassembler" that prints out the bytecode you produced in a human-readable form, as described below. The final result of your efforts should be a program `toy.cpp`, `toy.py` or `toy.java` that takes a single argument, the name of a TOY file. If given, for example, the input file `input.toy`, it should produce an output file `input.byte` containing the compiled bytecode. Next, I will break down the assignment for you. First, I will detail the execution model of the TOY virtual machine. Then I'll give you some advice on how to go about translating a TOY program into bytecode. (Note, this is only advice. You can do this whichever way works for you as long as you produce correct bytecode.) At the end of this assignment sheet, I'll give a specification of the virtual machine's bytecode.

1 Execution Model

The first two parts of TOY's execution state is made up of two stacks, the *data stack* and the *call stack*. The data stack is manipulated explicitly using TOY's built-in functions as discussed in the language description on the course website. The call stack is used to manage recursive calls as discussed in class. However, the call stack does not hold any stack frames because, as we will see, local variables need to be managed separately. Thus, the call stack only stores return addresses for the currently active function calls. The call stack is not explicitly exposed to the TOY program and is managed internally by the VM. All you need to know about the call stack is that calling a function using the `LOC` or `CBI` machine instruction pushes the current address on the stack and jumps to the beginning of the subroutine identified in the instruction. Executing `RET` inside the called subroutine returns to the address on the top of the call stack.

Since lambda expressions and references to functions defined using the `fun` keyword can be pushed onto the data stack, their lifetime may extend beyond the duration of the function call that created these function references. Thus, the referencing environment associated with the referenced functions must be preserved across the boundaries of function calls. This is exactly the situation we discussed in class that calls for the management of frames and closures. Thus, every reference to a function is accompanied by a reference to an associated frame to form a closure. The `LOC` instruction does in fact expect a closure as its argument and invokes the function in this closure with the frame included in the closure as the active frame. As discussed in class, each function call should start by creating a new frame whose parent is the currently active frame. This new frame holds all the local variables of the called function. The frames are managed by the VM's garbage collector, so frames only need to be created but do not need to be destroyed explicitly. In particular, there is no VM instruction to destroy frames.

The local variables in frames are referenced fairly easily. Instead of worrying about the exact size of the representation of each local variable, each frame simply has a number of slots. For example, the frame of a function with 3 local variables has 3 slots, one per variable. Accesses to these local variables are made by slot number. Similarly fields of user-defined types are identified by index within the object.

2 Semantic Analysis: The Abstract Syntax Tree

The first step towards building the compiler is to build an abstract syntax tree that reflects the nesting of lexical scopes. As you'll see below, the bytecode also needs a global table mapping various IDs

representing strings, symbols, and identifiers at runtime to their corresponding string representations. So you also need to produce two tables that stores these strings. We store string literals, symbol names, variable names, and the field names of user-defined types in a string table and type names in a separate type table. The reason is simple. Multiple occurrences of the same string literal or of a symbol whose name is also a variable name can all refer to the same string representation identified by a unique identifier. In contrast, two types with the same name defined in different scopes should be considered different types represented by different type IDs. To have a string representation of each type's name, we store this type name once per ID of a type with this name in the type table.

Let's use the two programs in Figures 1 and 2 to illustrate what your abstract syntax tree should look like. Their symbol tables and abstract syntax trees as produced by my reference implementation are shown in Figures 3 and 4.

At the beginning, the first part of the program representation consists of the string and type tables. The Fibonacci program contains no type definitions, so only the string table contains entries. The vector length program defines the type `vector` and its name is stored in the type table.

After these global tables comes the abstract syntax tree (AST). The root is the outer scope, a `Program`. Other types of scopes are `Function` and `Coroutine` (the latter only once we add coroutines to the language). Each scope stores the list of symbols (names) that were defined in this scope via type definitions, variable definitions, and local function or coroutine definitions. For each of these symbols, I store a list of associated attributes. The first one is its type (0 = variable, 1 = type, 2 = field read from a user-defined type, 3 = field write to a user-defined type). For a variable, I also store the number of its slot in the frame associated with the current scope and the index of the variable's name in the global string table. For a type, I store the type ID and its size. For a field read or write, I store the ID of the associated type, the index of the accessed field in a record of this type, and the index of the name of the accessor function in the global string table. For example, the vector type has type ID 0 and has two fields `x` and `y`. Thus, the information stored with `x` is `(2,0,0,?)` (2 = field read, 0 = type `vector`, 0 = position of `x` in a vector record). The information stored with `y=` is `(3,0,1,?)` (3 = field write, 0 = type `vector`, 1 = position of `y` in a vector record).

The list of statements of a scope lists the sequence of statements that occur in this scope. Every statement is of one of four types at this point. A local function defined as a named function or lambda is represented as a nested `Function` scope. Apart from that, we can have literals of type `Int`, `Float`, `|Char|`,`String` or `Symbol`; variable uses (any identifier that occurs as part of the scope's body); and reference uses (any reference that occurs as part of a scope's body). Note that named function definitions are split into a `Function` scope that defines this function followed by a `Var` node that defines a local variable with this function's name. We will translate this into assigning a closure for the function to the local variable in the final code. (Remember I mentioned that `fun foo ... .` is just syntactic sugar for `[ ... ] var foo`?) Strings and symbols are already translated into nodes of the form `String x` and `Symbol x` to refer to the xth string literal or symbol in the global string table.

3 Figuring Out What Identifiers Mean

The next pass over the AST transforms it into a different AST where `VarUse` and `RefUse` nodes are transformed into specialized node types that reflect what these nodes mean. We start at the scope containing the `VarUse`/`RefUse` and walk up the chain of nested scopes to find the closest scope that contains a local definition with this name. If such a scope is found, we check the type of the found symbol. If it's a variable, we turn it into a `LoadOrCall` node with attributes equal to the number of

hops we took to find the scope where this symbol resides. This is used at runtime to determine the number of static links to follow to find the relevant frame. The second attribute is the slot in that frame where the accessed variable is stored. `LoadOrCall` means that we need to translate this access into a `LOC` operation which copies the variable content to the stack if it's not a function closure and call the function closure if it is. For a reference to a variable, the node is called `Load` because it copies the variable content to the stack no matter what type the variable content has. If the found symbol is a type name, we create a `ObjCons` or `ObjConsRef` node to later generate an instruction that creates an object of the given type and the given size or create a function reference on the data stack that can later be called to create such an object. Similarly, field reads/writes are translated into `ObjRead`, `ObjWrite`, `ObjReadRef` or `ObjWriteRef` nodes.

If no symbol with the given name is found in any of the ancestor scopes of the current scope, the name may refer to a built-in function. Thus, I check the list of built-in functions and generate a `BuiltinCall` or `BuiltinRef` node if a built-in function with the given name is found. Otherwise, I raise a syntax error.

The result for the two programs in Figures 1 and 2 are shown in Figures 5 and 6.

4 Generating the Code

The final step is the generation of the code. This involves traversing the abstract syntax tree to collect all scopes, allocating a contiguous memory region for each scope, and populating the region allocated to each scope with the virtual machine instructions that implement the statements in this scope. Each inner scope represented by a child of type `Function` of the current scope needs to be translated into an instruction that creates a closure pointing to the current frame and to the address in the code block where the this inner function is stored. The end result is shown in Figures 7, 8, 9, and 10 in both byte code and virtual machine assembly language.

5 Virtual Machine Reference

5.1 File Structure

The code is to be provided to the virtual machine as a bytecode file. This file is divided into four sections: a *string block*, a *type block*, and a *code block*.

String block. The string block stores all string literals, variable names, symbol names, and field accessor names that occur in the program. The first 2 bytes of the string block store the number of strings stored in the string block. The remainder of the string block is the concatenation of the strings. Each string is terminated by a `'\0'` character.

Type block. The structure of the type block is identical to that of the string block. It stores the names of all user-defined types in the program.

Code block. The code block starts right after the type block. It is divided into a preamble followed by the code for all functions in the program. The preamble contains a number of `MCL` and `VAR` instructions to initialize the global scope storing all top-level functions. The preamble ends with a single `LOJ` instruction to the `main` function.

5.2 Machine Instructions

The instruction set of the virtual machine is limited. Each instruction is identified by a 1-byte opcode identifying the instruction. Depending on the instruction, this opcode is followed by 1–4 bytes holding the arguments of the instruction.

5.2.1 Handling Literal Data

`INT x` (`0x00`): **Integer literal.** Push an integer on the data stack. The next 4 bytes following this instruction encode the integer to be pushed.

`FLT x` (`0x01`): **Float literal.** Push a single-precision floating point number on the data stack. The next 4 bytes following this instruction encode the floating point number to be pushed.

`CHR x` (`0x02`): **Character literal.** Push a character literal on the data stack. The next byte is the ASCII code of the character to be pushed.

`STR x` (`0x03`): **String literal.** Push a string literal on the data stack. The next two bytes are the ID of the string in the string block to be pushed.

`SYM x` (`0x04`): **Symbol.** Push a symbol on the data stack. The next two bytes are the ID of the symbol to be pushed on the stack.

5.2.2 Manipulating Frames and Calling Functions

`MFR` (`0x10`): **Make frame.** Make a new frame whose parent is the currently active frame. The new frame becomes the new active frame. The next byte is the number of slots in the frame to be created.

`STO` (`0x11`): **Store.** Store the topmost element on the data stack in a slot of the current stack frame and remove it from the data stack. The next byte is the number of the slot where the element is to be stored.

`LOA` (`0x12`): **Load.** Retrieve the content of a slot in some frame and push it onto the data stack. This is followed by 2 bytes. The first byte is the number of parent pointers to follow from the currently active frame to identify the frame that holds the slot to be accessed. The second byte is the number of the slot in this frame to be accessed.

`LOC` (`0x13`): **Load or call.** Retrieve the content of a slot in some frame. If this content is a function closure, call the function with the frame referenced in the closure as the active frame. If the content is not a function closure, push the content onto the data stack. The first 2 bytes that follow a `LOC` instruction have the same meaning as for a `LOA` instruction. The 2 bytes after that are the index in the global string table of the name of the called function.

`LOJ` (`0x14`): **Load or jump.** Retrieve the content of a slot in some frame. If this content is a function closure, jump to the function with the frame referenced in the closure as the active frame. If the content is not a function closure, push the content onto the data stack. The 4 bytes that follow a `LOJ` instruction have the same meaning as for a `LOC` instruction.

`RET` (`0x1F`)**: Return.** Return from the current function to its caller.

5.2.3 Creating Function Closures

`MCL` (`0x20`)**: Make closure.** Create a closure consisting of a function and the currently active frame. The next 4 bytes are the address of the function in the program's code block.

5.2.4 Accessing Built-in Function

`CBI` (`0x30`)**: Call built-in.** Call a built-in function. The next byte identifies the built-in function.

`LBI` (`0x31`)**: Load built-in.** Push a closure identifying a built-in function onto the data stack. The next byte identifies the built-in function.

5.2.5 Manipulating User-Defined Types

`MOB` (`0x40`)**: Make object.** Push a user-defined object onto the data stack. This is followed by 2 bytes. The first byte is the ID of the type of the object to be created. The second byte is the number of fields this type has.

`RDF` (`0x41`)**: Read field.** Pop the topmost element from the data stack. This must be an object of a user-defined type. Push a field of this object onto the stack. The instruction is followed by 4 bytes. The first byte is the ID of the user-defined type expected on the stack. The second byte is the index of the field to be read. The remaining two bytes are the index in the global string table of the name of the field accessor function.

`WRF` (`0x42`)**: Write field.** Expect two elements on the top of the data stack. The topmost element can be of any type. The one below must be of a user-defined type. Pop the topmost element from the data stack and store it in a field of the object below it. The instruction is followed by 4 bytes. The first byte is the ID of the user-defined type the second element from the top of the stack must have. The second byte is the index of the field to be written. The remaining two bytes are the index in the global string table of the name of the field accessor function.

5.3 Built-In Functions

The built-in functions available in TOY were discussed in the language description. Since these functions aren't defined in any frame in your program, you have to have a different way of calling them or putting a function closure referring to them on the stack. As discussed above, the VM has instructions `CBI` and `LBI` to do this. The following are the identifier codes of all built-in functions:

call	0	Float	33
?	1	Bool	34
??	2	Char	35
true	3	format	36
false	4	asInt	37
&&	5	asBool	38
||	6	asFloat	39
not	7	asSymbol	40
=	8	fromString	41
!=	9	getChar	42
<	10	putChar	43
>	11	getStr	44
<=	12	putStr	45
>=	13	nl	46
+	14	open	47
-	15	close	48
*	16	eof	49
/	17	getCharF	50
%	18	putCharF	51
&	19	getStrF	52
|	20	putStrF	53
array	21	nlF	54
push	22	getArgs	55
pop	23	dup	56
@	24	dupn	57
@=	25	drop	58
=0	26	dropn	59
size	27	rotl	60
resize	28	rotln	61
++	29	rotr	62
new	30	rotrn	63
ref	31	swap	64
Int	32	swapn	65

6 Submission Instructions

Put your scanner implementation into a single zip-file containing all code required to run it, including the scanner used to translate the input text into a token stream and any other supplementary files that may be needed. Email this file to Arash Kayhani (arash.kayhani@dal.ca) before midnight, Apr 9, 2018.

```
fun fibonacci // n -- F_n
  fun fib // n-i F_{i-1} F_i -- F_n
    rotl dup 0 =
    [ drop swap drop ]
    [ 1 - rotr dup rotl + fib ] ??
  .
  0 1 fib
.

// Calculate the 10th Fibonacci number and print it to stdout
fun main
  :start drop // Doing this only to show how to handle symbols
  10 fibonacci format "F_10 = " swap ++ putStr nl
.
```

Figure 1: A complete program computing the 10th Fibonacci number.

```
type vector x y .

fun sqrt // x -- sqrt(x)
  var x
  fun go
    dup dup x swap / + 2 /
    dup rotl != %go ?
  .
  1 go
.

fun vector_len // vector(x, y) -- sqrt(x * x + y * y)
  dup x dup * swap y dup * + sqrt
.

// Create the vector (3,4), calculate its length and print it to stdout.
fun main
  vector 3 x= 4 y= vector_len format "len(3,4) = " swap ++ putStr nl
.
```

Figure 2: A complete program computing the length of a vector.

```
Strings:
  fibonacci
  fib
  main
  start
  F_10 =
Types:
Parse tree:
Program
  Symbols: fibonacci((0, 0, 0)) main((0, 1, 2))
  Statements:
    Function
      Symbols: fib((0, 0, 1))
      Statements:
        Function
          Symbols:
          Statements:
            VarUse = rotl
            VarUse = dup
            Int = 0
            VarUse = =
            Function
              Symbols:
              Statements:
                VarUse = drop
                VarUse = swap
                VarUse = drop
            Function
              Symbols:
              Statements:
                Int = 1
                VarUse = -
                VarUse = rotr
                VarUse = dup
                VarUse = rotl
                VarUse = +
                VarUse = fib
            VarUse = ??
        Var = fib
        Int = 0
        Int = 1
        VarUse = fib
    Var = fibonacci
    Function
      Symbols:
      Statements:
        Symbol = 3
        VarUse = drop
        Int = 10
        VarUse = fibonacci
        VarUse = format
        String = 4
        VarUse = swap
        VarUse = ++
        VarUse = putStr
        VarUse = nl
    Var = main
```

Figure 3: The symbol table and abstract syntax tree for the Fibonacci program.

```
Strings:
  x
  x=
  y
  y=
  sqrt
  go
  vector_len
  main
  len(3,4) =
Types:
  vector
Parse tree:
Program
  Symbols: vector((1, 0, 2)) x((2, 0, 0, 0)) x=((3, 0, 0, 1)) y((2, 0, 1, 2)) y=((3, 0, 1, 3))
           sqrt((0, 0, 4)) vector_len((0, 1, 6)) main((0, 2, 7))
  Statements:
    Function
      Symbols: x((0, 0)) go((0, 1, 5))
      Statements:
        Var = x
        Function
          Symbols:
          Statements:
            VarUse = dup
            VarUse = dup
            VarUse = x
            VarUse = swap
            VarUse = /
            VarUse = +
            Int = 2
            VarUse = /
            VarUse = dup
            VarUse = rotl
            VarUse = !=
            RefUse = go
            VarUse = ?
        Var = go
        Int = 1
        VarUse = go
    Var = sqrt
    Function
      Symbols:
      Statements:
        VarUse = dup
        VarUse = x
        VarUse = dup
        VarUse = *
        VarUse = swap
        VarUse = y
        VarUse = dup
        VarUse = *
        VarUse = +
        VarUse = sqrt
    Var = vector_len
    Function
      Symbols:
      Statements:
        VarUse = vector
        Int = 3
        VarUse = x=
        Int = 4
        VarUse = y=
        VarUse = vector_len
        VarUse = format
        String = 8
        VarUse = swap
        VarUse = ++
        VarUse = putStr
        VarUse = nl
    Var = main
```

Figure 4: The symbol table and abstract syntax tree for the vector length program.

```
Strings:
  fibonacci
  fib
  main
  start
  F_10 =
Types:
Parse tree:
Program
  Symbols: (0, 0, 0) (0, 1, 2)
  Statements:
    Function
      Symbols: (0, 0, 1)
      Statements:
        Function
          Symbols:
          Statements:
            BuiltinCall = 60
            BuiltinCall = 56
            Int = 0
            BuiltinCall = 8
            Function
              Symbols:
              Statements:
                BuiltinCall = 58
                BuiltinCall = 64
                BuiltinCall = 58
            Function
              Symbols:
              Statements:
                Int = 1
                BuiltinCall = 15
                BuiltinCall = 62
                BuiltinCall = 56
                BuiltinCall = 60
                BuiltinCall = 14
                LoadOrCall = 2,0,1
            BuiltinCall = 2
        Var = 0
        Int = 0
        Int = 1
        LoadOrCall = 0,0,1
    Var = 0
    Function
      Symbols:
      Statements:
        Symbol = 3
        BuiltinCall = 58
        Int = 10
        LoadOrCall = 1,0,0
        BuiltinCall = 36
        String = 4
        BuiltinCall = 64
        BuiltinCall = 29
        BuiltinCall = 45
        BuiltinCall = 46
    Var = 1
```

Figure 5: The symbol table and abstract syntax tree for the Fibonacci program, with meanings of variable and reference uses resolved.

```
Strings:
  x
  x=
  y
  y=
  sqrt
  go
  vector_len
  main
  len(3,4) =
Types:
  vector
Parse tree:
Program
  Symbols: (1, 0, 2) (2, 0, 0, 0) (3, 0, 0, 1) (2, 0, 1, 2) (3, 0, 1, 3) (0, 0, 4) (0, 1, 6) (0, 2, 7)
  Statements:
    Function
      Symbols: (0, 0, 0) (0, 1, 5)
      Statements:
        Var = 0
        Function
          Symbols:
          Statements:
            BuiltinCall = 56
            BuiltinCall = 56
            LoadOrCall = 1,0,0
            BuiltinCall = 64
            BuiltinCall = 17
            BuiltinCall = 14
            Int = 2
            BuiltinCall = 17
            BuiltinCall = 56
            BuiltinCall = 60
            BuiltinCall = 9
            Load = 1,1,5
            BuiltinCall = 1
        Var = 1
        Int = 1
        LoadOrCall = 0,1,5
    Var = 0
    Function
      Symbols:
      Statements:
        BuiltinCall = 56
        ObjectRead = 0,0,0
        BuiltinCall = 56
        BuiltinCall = 16
        BuiltinCall = 64
        ObjectRead = 0,1,2
        BuiltinCall = 56
        BuiltinCall = 16
        BuiltinCall = 14
        LoadOrCall = 1,0,4
    Var = 1
    Function
      Symbols:
      Statements:
        ObjectCons = 0,2
        Int = 3
        ObjectWrite = 0,0,1
        Int = 4
        ObjectWrite = 0,1,3
        LoadOrCall = 1,1,6
        BuiltinCall = 36
        String = 8
        BuiltinCall = 64
        BuiltinCall = 29
        BuiltinCall = 45
        BuiltinCall = 46
    Var = 2
```

Figure 6: The symbol table and abstract syntax tree for the vector length program, with meanings of variable and reference uses resolved.

```
00000000: 0005 6669 626f 6e61 6363 6900 6669 6200  ..fibonacci.fib.
00000010: 6d61 696e 0073 7461 7274 0046 5f31 3020  main.start.F_10
00000020: 3d20 0000 0010 0220 0000 0015 1100 2000  = ..... ...... .
00000030: 0000 6811 0114 0001 0002 1001 2000 0000  ..h......... ...
00000040: 2e11 0000 0000 0000 0000 0000 0113 0000  ................
00000050: 0001 1f10 0030 3c30 3800 0000 0000 3008  .....0<08.....0.
00000060: 2000 0000 4820 0000 0051 3002 1f10 0030   ...H ...Q0....0
00000070: 3a30 4030 3a1f 1000 0000 0000 0130 0f30  :0@0:........0.0
00000080: 3e30 3830 3c30 0e13 0200 0001 1f10 0004  >080<0..........
00000090: 0003 303a 0000 0000 0a13 0100 0000 3024  ..0:..........0$
000000a0: 0300 0430 4030 1d30 2d30 2e1f            ...0@0.0-0..
```

Figure 7: The bytecode of the Fibonacci program in binary form.

```
==STRINGS==
0: 'fibonacci'
1: 'fib'
2: 'main'
3: 'start'
4: 'F_10 = '
==CODE==
00000000: MFR 2
00000002: MCL 0x15
00000007: STO 0
00000009: MCL 0x68
0000000e: STO 1
00000010: LOJ 0,1,2
00000015: MFR 1
00000017: MCL 0x2e
0000001c: STO 0
0000001e: INT 0
00000023: INT 1
00000028: LOC 0,0,1
0000002d: RET
0000002e: MFR 0
00000030: CBI 60
00000032: CBI 56
00000034: INT 0
00000039: CBI 8
0000003b: MCL 0x48
00000040: MCL 0x51
00000045: CBI 2
00000047: RET
00000048: MFR 0
0000004a: CBI 58
0000004c: CBI 64
0000004e: CBI 58
00000050: RET
00000051: MFR 0
00000053: INT 1
00000058: CBI 15
0000005a: CBI 62
0000005c: CBI 56
0000005e: CBI 60
00000060: CBI 14
00000062: LOC 2,0,1
00000067: RET
00000068: MFR 0
0000006a: SYM 3
0000006d: CBI 58
0000006f: INT 10
00000074: LOC 1,0,0
00000079: CBI 36
0000007b: STR 4
0000007e: CBI 64
00000080: CBI 29
00000082: CBI 45
00000084: CBI 46
00000086: RET
```

Figure 8: The bytecode of the Fibonacci program in virtual machine assembly.

```
00000000: 0009 7800 783d 0079 0079 3d00 7371 7274  ..x.x=.y.y=.sqrt
00000010: 0067 6f00 7665 6374 6f72 5f6c 656e 006d  .go.vector_len.m
00000020: 6169 6e00 6c65 6e28 332c 3429 203d 2000  ain.len(3,4) = .
00000030: 0001 7665 6374 6f72 0010 0320 0000 001c  ..vector... ....
00000040: 1100 2000 0000 5611 0120 0000 0076 1102  .. ...V.. ...v..
00000050: 1400 0200 0710 0211 0020 0000 0032 1101  ......... ...2..
00000060: 013f 8000 0013 0001 0005 1f10 0030 3830  .............080
00000070: 3813 0100 0000 3040 3011 300e 0140 0000  8.....0@0.0.....
00000080: 0030 1130 3830 3c30 0912 0101 3001 1f10  .0.080<0....0...
00000090: 0030 3841 0000 0000 3038 3010 3040 4100  .08A....080.0@A.
000000a0: 0100 0230 3830 1030 4013 0100 0004 1f10  ...080.0@.......
000000b0: 0040 0002 0140 4000 0042 0000 0001 0140  .@.......B......
000000c0: 8000 0042 0001 0003 1301 0100 0630 2403  ...B.........0$.
000000d0: 0008 3040 301d 302d 302e 1f              ..0@0.0-0..
```

Figure 9: The bytecode of the vector length program in binary form.

```
==STRINGS==
0: 'x'
1: 'x='
2: 'y'
3: 'y='
4: 'sqrt'
5: 'go'
6: 'vector_len'
7: 'main'
8: 'len(3,4) = '
==TYPES==
0: 'vector'
==CODE==
00000000: MFR 3
00000002: MCL 0x1c
00000007: STO 0
00000009: MCL 0x56
0000000e: STO 1
00000010: MCL 0x76
00000015: STO 2
00000017: LOJ 0,2,7
0000001c: MFR 2
0000001e: STO 0
00000020: MCL 0x32
00000025: STO 1
00000027: FLT 1.0
0000002c: LOC 0,1,5
00000031: RET
00000032: MFR 0
00000034: CBI 56
00000036: CBI 56
00000038: LOC 1,0,0
0000003d: CBI 64
0000003f: CBI 17
00000041: CBI 14
00000043: FLT 2.0
00000048: CBI 17
0000004a: CBI 56
0000004c: CBI 60
0000004e: CBI 9
00000050: LOA 1,1
00000053: CBI 1
00000055: RET
00000056: MFR 0
00000058: CBI 56
0000005a: RDF 0,0,0
0000005f: CBI 56
00000061: CBI 16
00000063: CBI 64
00000065: RDF 0,1,2
0000006a: CBI 56
0000006c: CBI 16
0000006e: CBI 64
00000070: LOC 1,0,4
00000075: RET
00000076: MFR 0
00000078: MOB 0,2
0000007b: FLT 3.0
00000080: WRF 0,0,1
00000085: FLT 4.0
0000008a: WRF 0,1,3
0000008f: LOC 1,1,6
00000094: CBI 36
00000096: STR 8
00000099: CBI 64
0000009b: CBI 29
0000009d: CBI 45
0000009f: CBI 46
000000a1: RET
```

Figure 10: The bytecode of the vector length program in virtual machine assembly.