Assignment 3: Virtual Machine (VM)

Ali Zartash

1 Design

This assignment involved creating a virtual machine for the MITScript language. It is a follow up to Assignment 2 which involved creating a recursive interpreter for MITScript that runs the MITScript code directly in C++. The VM was an improvement on this – it involves a Compiler which converts the MITScript code to an intermediate Representation, and then a simple non-recursive bytecode Interpreter that runs the bytecode.

1.1 Compiler

The first part of the VM is the compiler that translates the source code into intermediate bytecode representation. Everything in the intermediate bytecode representation is within a "main" function that contains other hyper-parameters like constants used, local variable names, variables references in child scopes, variables from parent scopes. The function then also contains instructions which define a linear control flow ("goto" and "if" statements allow jumps). The syntax and semantics of the bytecode was given to us by the 6.S081 staff and it was easy to represent the function as a class which – the same as other data types like records, strings, integers, and Booleans – were all subclasses of a Value data type. This is similar to the recursive interpreter from Assignment 2 and makes sense because all these data types are first-class values in the dynamic MITScript Language. I also include a closure data type which is a function with variable references used as well as a reference data type which is a reference to any Value. These are required by the VM semantics/syntax.

The way the compiler generates code is very similar to how the recursive interpreter from assignment 2 worked. The compiler is a Visitor object and uses dual dispatch to visit the AST generated by the MITScript parser. It then generates from the AST, a control flow graph. The way I am implementing control flow graph is via a linked graph – most of the graph would be a linked list but there are some (while, if) blocks that branch. The nodes just contain pointers to neighbor nodes as well as Instructions. The compiler then iterates through this CFG and packs the instructions into a bytecode Function (main). Another class also used is the FuncBB class which is part of the CFG representation and represents the CFG for a function. It is basically a wrapper around the CFG making the implementation easier.

I do not use a symbol table explicitly. Rather the FuncBB class stores all the symbols (as well as the CFG) and acts as a type of symbol table. I think this was a bad decision and made my code much more complicated than would have been with a symbol table – but this also works and too much of the compiler was dependent on this decision to change it.

1.2 Interpreter

After the bytecode Function (main) has been generated by the Compiler, the interpreter is actually very simple. It has a stack of frames, as well as a stack of Functions (they correspond with each other). The interpreter simply loops through the instructions in the current (top of the stack) Function working in the current (top of the stack) frame. Control flow statements like goto cause the interpreter to jump – thus it keeps a PC as well referring the number of the current instruction within the current function. Calling a function generates a new stack-frame and puts it on top of the stack, while also putting the called function on top of the function stack (PC is also zero now and has state-behavior so when the function returns the calling function is at the right place). The interpreter then just continues executing instructions.

Another key detail is the Operand Stack. This is like the working memory of the interpreter and all the operations, arguments, references, etc. are pushed and popped from this stack. I actually exposed the iterators of this stack just to print it for debugging or otherwise.

2 EXTRAS

Usage:

mitscript (file_path) -s | -sp | b [-mem N]

The s flag treats the input file as source and runs it.
The sp flag runs treats the input similarly but doesn't run the code but rather prints to cout the bytecode representation (the prettyprinter was provided by 6.S081 staff).
The b flag treats input as bytecode then parses it into a main Function and then runs it.
There is also a mitscriptc file which does the same thing as mitscript with the sp flag. (prefer mitscript with sp flag)
The mem flag is described in Assignment 4. It is optional.
All executables are put in a binaries directory.
The tests directory in the root has some scripts that automate testing of all components.

3 DIFFICULTIES

This was a difficult assignment. But I think all the public and private tests of Assignment 2 pass now. I also have a script which produces bytecode files and then runs the VM with the –b argument on these bytecode files and we get the same results.

The main difficulty was that it was a very large amount of code – especially for 1 person as I was doing this assignment implementation myself. It is also hard to debug because one cannot really test until both the interpreter and the compiler are almost completely implemented in which case errors may cause a large amount of code to be changed and re-written. But I think this is a general feature of writing compilers for languages such as MITScript, Python, etc., rather than an issue with the assignment. I had to examine the bytecode very often to see where the issues were. A positive side effect was that this forced me to gain a good understanding of the operation of the VM.

I think it was fun though and the difficulties were worth it. I also did the project on my own so had the opportunity to make all the design decisions.

4 CONTRIBUTIONS

This version of Assignment 3 has been done entirely by me (Ali Zartash). My teammates worked on a separate implementation – this is because our group is a group of 4 and I felt like I can learn more by implementing the whole VM (and GC in Assignment 4) by myself. This is also a good example of N-version programming so when we are optimizing for performance we can pick and choose the better/faster parts from either implementation.