Assignment 3: Writing a Feeny Bytecode Machine

Patrick S. Li

September 10, 2015

1 Introduction

In this assignment, you will implement the bytecode machine for running Feeny programs after being compiled to a bytecode format. The compiler for compiling Feeny source files into the bytecode format is provided for you. In the next exercise, you will write the bytecode compiler yourself.

Performance aside, separating the implementation of a programming language into a compiler and interpreter for a customized instruction set is often the simplest implementation strategy. AST interpreters are particularly easy to write when the semantics of the new language is similar to the semantics of the implementation language. But when that is not the case, a separate bytecode compiler and interpreter is a general technique that can be readily applied.

2 The Feeny Bytecode Machine and Language

Conceptually, the state of the Feeny bytecode machine is described by the following tables:

2.1 Machine State

The global variable map: A single name-to-value table holds the current value of all the global variables used in the program. Operations are supported for retrieving the value associated with a given name, and assigning a value to a given name.

The current local frame: The current local frame represents the context in which the current function is executing. It contains the values of the arguments to the function, the values of all local variables defined in the function, the address of the instruction that called the current function, and the local frame of the calling function (also known as the parent frame). In total, the local frame has (num_args + num_locals) *slots*. The first num_args slots contains the arguments to the function, and the remaining slots contains the local variables.

The operand stack: A single stack holds the temporary values of all intermediate results needed during the evaluation of a compound expression. It supports operations for pushing a value to the stack, popping a value from the stack, and peeking at the top value in the stack.

The instruction pointer: The instruction pointer contains the address of the instruction that will be executed next.

2.2 Bytecode Program Structure

The default implementation of `interpret_bc` simply prints out the bytecode program. Please take a look at the output and read the following description to understand the bytecode structure.

A Feeny bytecode program has the following structure:

1. **The constant pool**: A listing of program objects that can be referred to by their numerical index. There are six different types of program objects: Nulls, Ints, Strings, Slots, Methods, and Classes. See section 2.3 for a description of each of them.
2. **The global slots**: A listing of the global variables and functions in the program by their index in the constant pool. Each index is guaranteed to refer to either a Slot object or a Method object. A Slot object

indicates that the program uses a global variable with name described by the Slot object. A Method object indicates that the program uses a function with name, arguments, and body described by the Method object.

3. **The entry function**: The index to the Method object that represents the entry function of the program. The entry function is a zero argument function that is called upon starting interpretation of the program. The program halts when the entry function returns.

2.3 Program Objects

All objects in the constant pool fall into one of these six categories:

1. **Null objects**: Represents a null value in Feeny. Used for the Literal instruction.
2. **Int objects**: Represents a 32-bit integer. Used for the Literal instruction.
3. **String objects**: Represents a character string. Strings are used to represent names of functions, slots, methods, and labels, and are used in instructions related to those.
4. **Slot objects**: Contains a single index that refers to the String object representing the name of the slot. A Slot object can either represent a variable slot in an object, when it is referred to in a Class object, or it could represent a global variable, when it is referred to in the list of global slots.
5. **Method objects**: Represents either a method of an object, or a global function. It contains the index of the String object representing the name of the method, the number of arguments, the number of local variables, and a Vector (see `utils.c`) containing all the instructions in the method.
6. **Class objects**: Represents the object structure, the variable and method slots, for each type of object created by `object`. It contains a Vector containing indices to all the slots in the object. Each index

refers to either a Slot object, which represents a variable slot, or a Method object, which represents a method slot.

2.4 Machine Instructions

The Feeny machine accepts the following instructions:

2.4.1 Basic Instructions

- **Literal Instruction Lit(i)** : Retrieves the object at index i in the constant pool, which refers to either an Int object or a Null object, and pushes it onto the operand stack.
- **Array Instruction Array()** : First pops the initializing value from the operand stack, and then pops the length of the array from the operand stack. Creates a new array with the given length, with each element initialized to the given value, and pushes the array onto the operand stack.
- **Print Instruction Lit(format, n)** : Pops n values from the operand stack, and then prints them out according to the given format string. Arguments are spliced in from the deepest value in the stack (last popped) to the shallowest value in the stack (first popped). Null is then pushed onto the operand stack.
- **Set Local Instruction SetLocal(i)** : Sets the i'th slot in the current local frame to the top value in the operand stack.
- **Get Local Instruction GetLocal(i)** : Retrieves the i'th slot in the current local frame and pushes it onto the operand stack.
- **Set Global Instruction SetGlobal(i)** : Sets the global variable with name specified by the String object at index i to the top value in the operand stack.
- **Get Global Instruction GetGlobal(i)** : Retrieves the value of the global variable with name specified by the String object at index i, and pushes it onto the operand stack.

- **Drop Instruction Drop()** : Pops and discards the top value from the operand stack.

2.4.2 Object Instructions

- **Object Instruction Object(c)** : Retrieves the Class object at index c. Suppose that the class object contains n Slot objects, and m Method objects. This instruction will pop n values from the operand stack for use as the initial values of the variable slots in the object, and then an additional value for use as the parent of the object. The first variable slot is initialized to the deepest value on the stack (last popped) and the last variable slot is initialized to the shallowest value on the stack (first popped). A new object is created with the variable slots indicated by the Class object, initialized to the given values on the stack, with the method slots indicated by the Class object, and with the given parent object, and is pushed onto the operand stack.
- **Get Slot Instruction GetSlot(i)** : Pops a value from the operand stack assuming it is an object. Retrieves the value in the object stored at the variable slot with name given by the String object at index i, and pushes it onto the operand stack.
- **Set Slot Instruction SetSlot(i)** : Pops the value to store, x, from the operand stack, and then pops the object to store it into. Stores x into the object at the variable slot with name given by the String object at index i. x is then pushed onto the operand stack.
- **Call Slot Instruction CallSlot(i, n)** : Pops n values from the operand stack for the arguments to the call. Then pops the receiver object from the operand stack. The name of the method to call is given by the String object at index i. If the receiver is an integer or array, then the result of the method call (as specified by the semantics of Feeny) is pushed onto the operand stack. If the receiver is an object, then a new local frame is created for the context of the call. Slot 0 in the new local frame holds the receiver object, and the following n slots holds the argument values starting with the deepest value on the stack (last popped) and ending with the shallowest value on the stack (first popped). The new local frame has the current frame as its parent,

and the current instruction pointer as the return address. Execution proceeds by registering the newly created frame as the current frame, and setting the instruction pointer to the address of the body of the method.

2.4.3 Control Flow Instructions

- **Label Instruction Label(i)** : Associates a name with the address of this instruction. The name is given by the String object at index i.
- **Branch Instruction Branch(i)** : Pops a value from the operand stack. If this value is not Null, then sets the instruction pointer to the instruction address associated with the name given by the String object at index i.
- **Goto Instruction Goto(i)** : Sets the instruction pointer to the instruction address associated with the name given by the String object at index i.
- **Return Instruction Return()** : Registers the parent frame of the current local frame as the new current frame. Execution proceeds by setting the instruction pointer to the return address stored in the current local frame. The local frame is no longer used after a Return instruction, and any storage allocated for it may be reclaimed.
- **Call Instruction Call(i, n)** : Pops n values from the operand stack for the arguments to the call. The name of the function to call is given by the String object at index i. A new local frame is created for the context of the call. The first n slots in the frame holds the argument values starting with the deepest value on the stack (last popped) and ending with the shallowest value on the stack (first popped). The new local frame has the current frame as its parent, and the current instruction pointer as the return address. Execution proceeds by registering the newly created frame as the current frame, and setting the instruction pointer to the address of the body of the function.

3 Harness Infrastructure

A compiler is provided for you for compiling Feeny programs into the Feeny bytecode format. The function

```
void interpret_bc (Program* p)
```

in the file `src/vm.c` will be the entry point of your bytecode interpreter and will be called with the compiled bytecode.

3.1 Bytecode Data Structures

The data structures for representing the bytecode are defined in the file `src/bytecode.h`. `src/bytecode.c` implements a pretty printer for viewing the bytecode instructions as text. The data structure closely mirrors the description of the bytecode structure in section 2.4, and you may also read the implementation of the pretty printer to see how to operate on this data structure.

The `print_value` function prints out a constant in the constant pool. The `print_ins` function prints out a single bytecode instruction. The `print_prog` function prints out the entire bytecode program.

3.2 Test Harness

The provided bash script `run_tests` will compile the test programs in the `test` directory into bytecodes and run your bytecode interpreter on the result. To run it, type:

```
./run_tests vm.c
```

at the terminal. For each test program, e.g. `cplx.feeny`, the output of your bytecode interpreter will be stored in `output/cplx.out`. The default implementation of `interpret_bc` does nothing except print out the bytecode program.

Please ensure that your implementation can be driven correctly by `run_tests`. Your assignment will be graded using this script.

3.3 Compiling and Running Manually

To compile and run your implementation of the bytecode interpreter manually, you may also follow these steps. Compile your interpreter by typing:

```
gcc src/cfeeny.c src/utils.c src/bytecode.c src/vm.c
   -o cfeeny -Wno-int-to-void-pointer-cast
```

at the terminal. This will create the `cfeeny` executable.

Next, you have to run the supplied compiler which will read in the source text for a Feeny program and compile it to a binary file containing the bytecode instructions.

```
./compiler -i tests/cplx.feeny -o output/cplx.bc
```

Finally, call the `cfeeny` executable with the binary bytecode file as its argument to start the interpreter.

```
./cfeeny output/cplx.bc
```

4 Report

Implement your bytecode interpreter and place it in the `src` directory, naming it `vm_xx.c`, where `xx` is replaced with your initials. Then ensure that it can be properly driven by the testing harness by typing:

```
./run_tests vm_xx.c
```

Include your implementation of Towers of Hanoi, Stacks, and More Towers of Hanoi from exercise 1, and ensure that your interpreter works correctly for those as well.

1. (66 points) **Interpreter Statistics** : Create a table in your report that measures the following statistics for the test programs: `bsearch.feeny`, `inheritance.feeny`, `cplx.feeny`, `lists.feeny`, `vector.feeny`, `fibonacci.feeny`, `sudoku.feeny`, `sudoku2.feeny`, `hanoi.feeny`, `stacks.feeny`, `morehanoi.feeny`.
   (a) (1 point) Total amount of time (in milliseconds) for `interpret_bc` to run and return.

(b) (1 point) Total number of method calls.

(c) (1 point) Total number of method calls where receiver is an integer.

(d) (1 point) Total number of method calls where receiver is an array.

(e) (1 point) Total number of method calls where receiver is an object.

(f) (1 point) Total amount of time (in milliseconds) spent looking up an entry in an environment.

2. (20 points) **Bytecode Verification**: The Feeny machine is capable of a much richer set of behaviors than the Feeny language. For example, the Feeny language only allows functions to return a single value, whereas the Feeny machine can put as many values on the operand stack as it likes. It is actually *too* rich in the sense that many of these extra behaviours are undesirable, and may hinder opportunities for optimization later on. Consider writing a bytecode verifier that restricts the bytecode format to be wellformed and as strict as possible while still being to able to express all Feeny programs. What would the verifier check for?

3. (5 points) **Environment Lookup** : Does the bytecode interpreter spend more or less time looking up entries in environments as compared to the AST interpreter? What is the cause of this difference?

4. (5 points) **Control Flow**: Suppose we wanted to extend the Feeny language with support for return, break, continue, and goto statements. Is the Feeny virtual machine expressive enough to support these constructs as it is? What additional instructions are necessary?

5. (4 points) **Control Flow Implementation**: Consider extending the Feeny language with a return statement. Contrast the difference in implementation strategies between adding return statements to the AST interpreter versus the bytecode interpreter. Which one requires more support from the implementation language?

5 Deliverables

Students may work in pairs or alone. Please submit your answers to section 4 as *report_XX.pdf*, and your programs as *vm_xx.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign3_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny3]` in the subject header.