Assignment 4: Writing a Feeny Bytecode Compiler

Patrick S. Li

September 18, 2015

1 Introduction

In this assignment, you will implement the bytecode compiler for compiling a Feeny program expressed as an abstract syntax tree into the Feeny bytecode format. After implementing this portion of the system, you will have a complete self-contained system that is able to compile and execute Feeny programs.

While the compiler is a separate part of the system that does not influence the design or implementation of the virtual machine, it is important to realize that virtual machines are seldom developed in isolation. Although in our exercises you implemented the bytecode interpreter and will now implement the compiler, in reality the two pieces would likely be developed in tandem. The design of the compiler will influence the bytecode IR, and the interpreter will also influence the design of the compiler.

2 Harness Infrastructure

Your task is to compile a Feeny program from its abstract syntax tree into the Feeny bytecode IR. The semantics of the Feeny language and of the Feeny bytecode are defined in the previous assignments.

A minimal framework is provided for you for lexing and parsing Feeny programs into in-memory datastructures that represent the abstract syntax tree.

The function

```
Program* compile (ScopeStmt* stmt)
```

in the file `src/compiler.c` will be the entry point of your bytecode compiler, and will be called with the AST datastructure.

As in the last assignment

```
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 result of your bytecode compiler.

2.1 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. You may also read the implementation of this 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.

The compiler will need to correctly create these data structures as its output, so that they can be read back by your interpreter.

2.2 Test Harness

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

```
./run_tests compiler.c 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 implementations of `compile` and `interpret_bc` do nothing except print out the AST and bytecode IR respectively.

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

2.3 Compiling and Running Manually

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

```
gcc -std=c99 src/cfeeny.c src/utils.c src/ast.c
    src/bytecode.c src/compiler.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 parser which will read in the source text for a Feeny program and dump it to a binary AST file.

```
./parser -i tests/cplx.feeny -oast output/cplx.ast
```

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

```
./cfeeny output/cplx.ast
```

3 Report

Implement your bytecode compiler and interpreter and place it in the `src` directory, naming it `compiler_xx.c` and `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 compiler_xx.c vm_xx.c
```

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

1. (55 points) **Compiler Statistics**: Create a table in your report that measure and calculate 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) (2 point) Total amount of time (in milliseconds) for `compile` to run and return.
   (b) (2 point) Total amount of time (in milliseconds) for `interpret_bc` to run and return.
   (c) (1 point) Percentage of total time spent in compilation.
2. (20 points) **Reverse Engineering**: One supposed benefit of distributing programs in compiled form over source form is increased control over intellectual property. What information is lost about a Feeny program through compilation? Suppose we did our best to write a Feeny reverse compiler to turn Feeny bytecode IR back into source text. Provide an example of what `vector.feeny` might look like after reverse compilation.
3. (10 points) **Semantics**: Due to the design of the Feeny bytecode IR, the semantics of the Feeny AST interpreter is ever so slightly different than that of the Feeny bytecode interpreter. What is this difference? If Feeny were to become a widely used and critically acclaimed programming language, how might this difference lead to portability issues between different Feeny implementations?
4. (8 points) **Bytecode Specification**: Currently our system reads in a Feeny AST, compiles it into bytecode, and then directly interprets the bytecode. Thus the bytecode IR is completely internal to the system and not exposed to the user. If we want to be able to distribute Feeny programs in the compiled bytecode format, what else do we need to specify?
5. (7 points) **Labels**: To express control flow constructs, the Feeny bytecode IR uses labels to indicate targets of goto and branch instructions. This contrasts with the Java bytecode, which uses integer offsets. Discuss the advantages and disadvantages of the two designs.

4 Deliverables

Students may work in pairs or alone. Please submit your answers to section 3 as *report_XX.pdf*, and your programs as *compiler_XX.c*, *vm_XX.c*, *hanoi_XX.feeny*, *stacks_XX.feeny*, and *morehanoi_XX.feeny*. Zip all files together in a file called *assign4_XX.zip*. Replace *XX* above with your initials. Mail the zip file to `patrickli.2001@gmail.com` with `[Feeny4]` in the subject header.