

CS 577 Homework 1 – Extending a bytecode interpreter – due 10am, Thursday, April 10, 2008

On the course web page, you'll find C code for an interpreter for a small subset of JVM instructions. The Java subset supported includes integer arithmetic (except division and remainder), static methods, and a minimal set of output facilities. The interpreter is defined by a set of C files (`interp0.c`, `class.[ch]`, `basics.[ch]`, `bytecode.h`) and a `makefile`, which generates an executable `interp0`. This can be invoked just like the usual `java` interpreter on a single class file, but fails on programs outside the supported subset. There is also a small coverage test suite in the directory `tests` and a simple test driver shell script `dotest`.

Your primary assignment is to extend `interp0` to deal with integer arrays. This will involve adding support for about twelve new JVM instructions, and providing coverage tests for your modifications. There are also two other questions for you to answer about the behavior of `interp0`.

Details

The subset handled by the existing interpreter should be sufficient to execute simple integer programs involving `static` methods within a single class. (I may have left out one or two rarely generated instructions; if so, feel free to implement them!) The only way to do output (supported by a nasty special-case hack) is via `System.out.print(x)` where x is an integer or a string literal. Still, this is enough to write test cases that display their results, and will run under the ordinary JVM as well as under this interpreter. The interpreter dies with an “unimplemented” message about any instruction it can't cope with.

To handle arrays, you'll need to add cases to handle these additional instructions: `ACONST_NULL`, `ALOAD` (and its variants), `ARETURN`, `ARRAYLENGTH`, `ASTORE` (and its variants), `ILOAD`, `IASTORE`, `IF_ACMPEQ`, `IF_ACMPEQ`, `IFNONNULL`, `IFNULL`, and `NEWARRAY`. Most of the necessary code can be copied—or even reused without copying by making judicious use of fall-through `switch` cases—from the existing support for integers. The interesting cases are `NEWARRAY`, `ARRAYLENGTH`, `ILOAD`, and `IASTORE`. To see how these instructions are used by real Java programs, write `.java testcases` and use `javap -c` to examine the corresponding bytecode. Make sure you write test cases that exercise all these instructions!

You only need to implement `NEWARRAY` for `int` array elements. Represent an n -element array by a $(n + 1)$ -word heap block, with the length stored in the first word. To allocate a heap block, use the `heap_alloc` function defined in `runtime`. For now, don't worry about deallocation or garbage collection!

You need to arrange to “raise” the following built-in exceptions when appropriate:

```
NullPointerException
ArrayIndexOutOfBoundsException
NegativeArraySize
```

Do this using the `exception` function defined in `runtime`; the necessary `exn` values and corresponding strings are already defined for you. Don't try to implement real exceptions or exception handling! Make sure you write test cases that provoke all these exceptions, as well as the `OutOfMemory` exception (which can be raised by `heap_alloc`).

You must submit your test suite as well as your revised interpreter. Your test suite should cover all paths through the code you have added; i.e., it should cause each new instruction to be executed and each possible exception to be raised. Note that nearly all your new tests can be placed into a single file, but tests that raise exceptions need to go in separate files. Also, of course, the existing test suite should continue to run!

The easiest way to run tests is by using the `dotest` script, which stores standard output and error into `.out` and `.err` files, respectively, and compares them to any previously-established baseline files `.out.bak` and `.err.bak`. Your submitted test suite should include the `.out.bak` and `.err.bak` files so that I'll know what you think the output of the tests should be.

Additional Questions

In addition to the modifications requested above, please answer the following questions:

1. When evaluating a call to a static function (i.e., interpreting `INVOKESTATIC` instructions), the interpreter doesn't seem to do anything explicit about passing arguments. Yet somehow, the arguments do get passed. Explain how. Drawing a (rough) diagram may be quite helpful!
2. The interpreter actually uses two stacks. The data stack is explicitly managed by the interpreter, which raises an exception if the data stack overflows. But the control stack is implicit; it is actually the underlying C stack. What happens if the control stack overflows?

How to submit your homework.

Submit the homework by mail to `apt@cs.pdx.edu` with subject line "CS577 HW1." The elements of your submission should be plain-text attachments to your mail, either individual files or a single zip file. Submit:

- A file `interp1.c` containing your modified version of `interp0.c`.
- A collection of `.java` files constituting a coverage test for the new instructions you have added, together with the corresponding baseline test outputs stored in `.out.bak` and `.err.bak` files.
- A file `answers.txt` containing your answers to the additional questions.