CSE 6341, Programming Project 3
Due Thursday, March 7, 11:59 pm (30 points)

The goal of this project is to build an interpreter for the language from Project 2 (with some minor changes to the language, as defined below). The semantics to be implemented was discussed when we considered the definition of operational semantics. Your implementation will use the code for AST building from Project 2, with minor changes to `Interpreter.java`. Assume that the input program successfully passes the typechecking defined in Project 2; you **do not** need to perform this typechecking in your implementation of Project 3.

Set up

```
cd /home/buckeye.8/6341/proj; mkdir p3; cd p3
wget web.cse.ohio-state.edu/~rountev.1/6341/project/p2.tar.gz
tar -xvzf p2.tar.gz --strip-components=1
make
./plan t1
```

Edit `Interpreter.java` to add the following new exit codes:

```
public static final int EXIT_UNINITIALIZED_VAR_ERROR = 3;
public static final int EXIT_DIV_BY_ZERO_ERROR = 4;
public static final int EXIT_FAILED_STDIN_READ = 5;
```

Restriction

To simplify the project, the following restriction will be imposed on all input programs: no two variables have the same name. This also applies to variables that are in different blocks: so, code such as **int z = ...; {** … **{ int z =** …**; } }** is never going to be part of a valid input program.

You **do not** have to check whether this restriction is satisfied by the input program: just assume that it is and implement your interpreter under this assumption.

Details

The semantics was described in class. A few additional details:

1) Do **not** print the program. Comment out astRoot.print in main. Comment out any other printing of the AST. Do **not** print the program state. Do **not** print any testing/debugging messages you used for yourself when developing the code.

2) The are only two cases when you should print something:

- Printing - case 1: at a print statement of the form **print** *<expr>*; evaluate the expression to a number and then print that number using System.out.println(…).
- Printing - case 2: if there is a run-time error, print an error message and exit with the correct exit code

3) You **must** catch run-time errors for "use of unintialized variable" and "division by zero" and then exit the interpreter with the corresponding error codes listed above.

4) **readint** and **readfloat** expressions should read from UNIX *stdin* and produce a value of the corresponding type. If this reading cannot be performed successfully, the interpreter should exit with error code `EXIT_FAILED_STDIN_READ`.

Implement this functionality using a `Scanner` object from `java.util`. At the very beginning of program execution create a single object `Scanner s = new Scanner(System.in)` and then call `s.hasNextLong()` and `s.nextLong()` (and similarly for `hasNextDouble()` and `nextDouble()`) whenever you need to evaluate a **readint** or **readfloat** expression. Create only one such object and use it for every evaluation of **reading** and **readfloat**.

When executing the interpreter from the command line, you can put the input data in some file `data_file` and then do `./plan code_file < data_file`

5) Type INT in our language should be implemented by a Java `long` type. Type FLOAT in our language should be implemented by a Java `double` type. If you need to represent both possibilities with a single Java type, you can use `java.lang.Number`, which is a common superclass of `java.lang.Long` and `java.lang.Double` (and those themselves are wrappers around primitive types `long` and `double`, respectively).

6) Since each variable declaration uses a unique name (due to the restriction described above), you can implement the state **σ** with one map from variable names to values. There is no need to use a tree of maps. At the beginning of execution, the map is empty.

7) The evaluation of Boolean operators **&&** and **||** should use short-circuit semantics.

8) Unlike the semantics described in class, here we will impose an order of evaluation for arithmetic operators **+**,**-**,*****,**/** and comparison operators **<**, **<=**, **>**, **>=**, **==**, **!=**. The first operand of such an operator **must** be evaluated first. This restriction makes the semantics for Project 3 deterministic. Without this restriction, non-determinism exists because of **reading** and **readfloat** expressions, which have side effects (e.g., consider **x + readint x**). The language discussed in class does not have side effects for expressions; thus, all evaluation orders produce the same result.

9) A natural implementation approach is to add to each expression class a method `evaluate` that takes as input a reference to the state, evaluates the expression, and returns the resulting value. Similarly, for each statement class you can use a method `execute` that takes as input a reference to the state, executes the statement, and as a result changes the state.

Testing

Write many test cases and test your interpreter with them. Submit at least 5 test cases with your submission. The test cases you submit will not affect your score for the project. Put them in the same location as the provided file t1 and name them t2, …

Submission

After completing your project, do

```
cd p3; make clean; cd ..
tar -cvzf p3.tar.gz p3
```

Then submit `p3.tar.gz` in Carmen.

General rules (copied from the course syllabus)

Your submissions must be uploaded via Carmen by midnight on the due date. The projects must compile and run on **stdlinux**. Some students prefer to implement the projects on a different machine, and then port them to stdlinux. If you decide to use a different machine, it is entirely your responsibility to make the code compile and run correctly on stdlinux before the deadline. In the past many students have tried to port to stdlinux too close to the deadline, leading to last-minute problems and missed deadlines.

Projects should be done independently. General high-level discussion of projects with other students in the class is allowed, but **you must do all design, programming, testing, and debugging independently.** Projects that show excessive similarities will be taken as evidence of cheating and dealt with accordingly. Code plagiarism tools may be used to detect cheating. See the syllabus under "Academic Integrity".

The projects are due by 11:59 pm on the due day. You can submit up to 24 hours after the deadline; if you do so, your score will be reduced by 10%. **ONLY THE LAST SUBMITTED VERSION WILL BE CONSIDERED. Triple-check carefully that you have submitted the correct version. If you submit the wrong version of your code, and you get a low score (or zero score), I will NOT consider resubmissions – the original low/zero score will be assigned WITHOUT DISCUSSION.**

If you submit more than 24 hours after the deadline, the submission will not be accepted. NO EXCEPTIONS TO THIS RULE WILL BE CONSIDERED. NO REQUESTS FOR RESUBMISSION WILL BE CONSIDERED. MAKE SURE YOU SUBMIT THE CORRECT CODE VERSION.

Read the project description **very carefully, several times, start-to-end.** If you need any clarifications, contact me immediately (do **not** wait until the last minute). **Test extensively.**

Accommodations for sickness and other special circumstances will be made based on university guidelines. Please contact me **ahead of time** to arrange for such accommodations.