CS 431 – Assignment 5 – Spring 2025

Due: Tuesday, May 13^{th} **before 8:00 AM**

Total points: 500

Learning outcomes

- Implement the bytecode generator for your Bantam Java compiler whose output must be executable on the Java Virtual Machine.

Preparation

- Re-read chapter 3 of the Bantam Java manual. You must be an expert on Bantam Java's semantics before you can generate code that correctly implements Bantam Java programs.
- Review your notes and slides on the JVM and code generation. You must know the JVM's architecture and instruction set very well before you can generate bytecodes that are accepted by the JVM's bytecode verifier and that produce the correct execution of the source program.
- Read this handout carefully right away, and again just before submitting your work. **It would be a shame to lose points for failing to follow the directions given below.**
- Download the file `a5.zip` from the Canvas site. It will expand into a directory called `a5` containing the Bantam Java compiler shell.
- Copy your lexical definitions from Assignment 1, your parser from Assignment 3, and your type-checker from Assignment 4 to their respective places in the directory structure that you downloaded. Note: If you did not find all static semantic errors for assignment 4, you may still be able to complete this assignment since I will only test it with syntactically correct files with no static semantic errors. However, you will need to at least produce a class tree with symbol tables, etc., since the code generator needs it.

Your task is to complete the code generator for the Bantam Java compiler. For full credit, your compiler must output executable bytecodes that produce correct results and that satisfy the additional constraints set out in this handout (e.g., presence of indentation and comments in your assembly language files, correct values for the stack and local-variable array sizes, etc.). All your programming will take place in the `codegenjvm` package.

First, you will create a `JVMCodeGenerator` class (in its own file) with a `generate` method that goes through all the user-defined classes in the Bantam Java source program and generates Jasmin assembly-language instructions in a separate `.j` file (one per class). The name of this Jasmin file (before the extension) must be equal to the name of the class. To this end, the `generate` method will create a `CodeGenVisitor` object and let it visit the AST sub-tree for the class to generate the corresponding code.

Second, you will create a `CodeGenVisitor` class (in its own file) that extends the pre-defined `Visitor` class. Your new visitor will perform all the work involved in visiting the AST nodes in the appropriate order and generating the correct bytecodes.

For each class, the visitor must:

- Output to the `.j` file the standard class header. Since Bantam Java supports the `clone` method in the `Object` class, you should make sure that each class implements the `Cloneable` interface, which is as simple as adding the following line at the end of your header: `.implements java/lang/Cloneable`
  Check out the jasmin documentation for a description of this and other directives.
- Split members of the class into fields and methods.
- Output the fields, which involves recursively visiting the initializer expressions (if any).
- Add a default constructor for the class, which must initialize each field, using either the initializer expression or a default value.
- Output the `main` method needed by the runtime system to start the execution. This method is always the same since it simply calls the Bantam Java main method on a newly created `Main` instance.
- Output assembly code for each method in the class.

Here is a description of the actions to be taken based on the type of AST node being visited:

Field Recall that all fields are protected in Bantam Java.

Method Output the method's signature line, including formals. Just in case the method calls the `clone` method, add this line just after the method header: `.throws java/lang/CloneNotSupportedException`
then visit the body's statements and output the method closing directive. Beware that, in the case of `void` methods, you may have to add a `return` statement before the closing directive, since this statement may be omitted in the source program.

Formal Output its descriptor and update the information needed for the local-variable array.

DeclStmt Output comment with the name of the local variable and its index in the local-variable array; output code for the initialization expression; and maintain the index and type of the variable in the environment.

VarExpr Look up the appropriate value and push it onto the stack; there are several cases to cover: `this`, `super`, `null`, a field name, a formal parameter, a local variable, `this.<field>`, or `<array>.length` expression.

ArrayExpr Similarly to the previous case, there are several cases to consider.

ConstIntExpr You must use the most "efficient" bytecode for the specific value of the constant.

AssignExpr and ArrayAssignExpr Output the variable name in a comment; visit the RHS; note that the LHS can be `this.<field>`, `field`, a method parameter, or a local variable.

DispatchExpr Output a comment; use `this` as default object, otherwise visit the reference expression, output code for the argument expressions (if any). Remember to use the appropriate opcode based on whether the method call is of the form `super.m()` or `m()`. Also, when a method is called on an array, it must be the `clone` method, which should be looked up in the `Object` class.

BinaryLogicOrExpr and BinaryLogicalAndExpr Make sure to implement short-circuit evaluation.

IfStmt Output comments before the predicate as well as the "then" and "else" blocks. Keep in mind that the predicate could be: a `BinaryCompExpr`, a `ConstBooleanExpr`, or a general `Expr`. As explained in class, the stack height must end up the same after each branch of the "if"; therefore, you may need to output a series of `pop`'s after one or the other block, or both. The objective is for the stack height to get back to the level it was at before the "if" statement.

WhileStmt and ForStmt Output comments before the Boolean expression (as well as the loop initialization and update expressions, if any) and the loop body. The same comments about the types of predicates and the stack height given for the "if" statement also apply here.

Generating Jasmin instructions for the remaining types of AST nodes should be straightforward, with the following caveats:

- When outputting string literals to the `.j` file, remember to process the special sub-strings "\n" and "\"" appropriately.
- When processing blocks, remember to enter and exit scopes accordingly, and to minimize the number of locations needed in the local-variable array.
- Use fully-qualified class names where appropriate.
- Use the following systematic scheme for naming labels: the first label for each method should be called "L1:", the next one "L2:", etc.
- All `visit` methods should return `null`.
- For each method, you'll have to determine the exact size of the local variable array and the size of the operand stack. Too small a value will cause the bytecode verifier to reject your class file. Too large a value would enable your program to run but would waste memory, and thus cost you points.
- You might want to keep track of the stack height after each instruction is output.
- You must also keep track of the types of formal parameters and local variables in the environment.

Feel free to use as many helper methods (and classes) as necessary to **keep your code modular**.

Finally, note that the code that you downloaded contains the runtime system (in the form of `.class` files) required by Bantam Java. In particular, the provided runtime system uses Java's `String` and `Object` classes, and contains the `.class` files (in the `lib` directory) for the `TextIO` and `Sys` classes.

Testing your compiler

To build your compiler, 'cd' to the top-level `a5` directory, and type: `ant src`; To run your compiler on a Bantam Java program, say `TicTacToe.btm`, in the `./tests` directory, 'cd' to this directory and type:

```
ant tic-tac-toe
```

which will produce a `tic-tac-toe.jar` file, which you can then run by typing:

```
java -jar tic-tac-toe.jar
```

In order to find the lowercase name (say, `tic-tac-toe`) corresponding to the name of a Bantam Java source file (say, `TicTacToe.btm`), look it up in the `build.xml` file.

Finally, to run your compiler on all files in the `./tests` directory, 'cd' to the `a5/tests` directory, and type: `ant`.

Your code generator is only expected to handle type-correct programs. If a syntactical, grammatical, or type error is detected, your compiler will stop with one or more errors, and no code generation will take place. Note that the file `ArraySizeTooLarge.btm` does NOT generate any runtime exception when run on the JVM.

One strategy for completing this assignment is to first decide what bytecode you want to generate for each AST node, then to create a Jasmin file by hand to test it and verify that it works as expected, and only then to have your compiler generate it automatically. Once that's done, check the `.j` file produced by your compiler and make sure it is correct. Finally, run it through Jasmin and

then through the JVM. If you are not sure whether the output of your compiled code is correct, simply transform the Bantam Java source into a valid Java program and check your output against the one produced by the Java program.

Submission procedure

Before the deadline, you must copy to the A5 dropbox on Canvas a single zip file called `a5.zip` containing your COMPLETE compiler, including your scanner, parser, semantic analyzer, and code generator. Make sure to type `ant clean` in the top-level `a5` directory before submitting it.

Make sure to format your code nicely, with proper indentation and no more than 80 characters to a line to **prevent wraparounds** in my editor.

Points will be deducted if your compiler does not work properly, and/or if you do not follow the naming/formatting/submission requirements listed in this handout.

The late-submission penalty stated in the syllabus applies to this assignment.

Good luck and have fun!