SE 421: Extra Credit Assignment #1

Due on November 26, 2018 at 12:00 PM (noon)

Instructor: Ben Holland

https://github.com/SE421/extra-credit-assignment1

Student Name:

Table 1: Brainf*ck Commands

Character	Meaning
>	increment the data pointer (to point to the next cell to the right).
<	decrement the data pointer (to point to the next cell to the left).
+	increment (increase by one) the byte at the data pointer.
-	decrement (decrease by one) the byte at the data pointer.
.	output the byte at the data pointer.
,	accept one byte of input, storing its value in the byte at the data pointer.
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command.
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

Problem 1

(15 points)

In this assignment you will be building a program analysis tool from scratch for the Brainf*ck language [1]. The Brainf*ck language consists of eight instructions described in Table 1. A Brainf*ck interpreter and debugger is available as an Eclipse plugin from the Eclipse Marketplace [2].

Your first task will be to build a parser to populate the abstract syntax tree (AST) of the Brainf*ck language. For this task we will be using ANTLR parser generator [3]. For a simple example of what ANTLR can be used to accomplish see `https://stackoverflow.com/a/1932664/475329`. ANLTR can be integrated into Eclipse (see [4]) as well as a variety of other tools. Note that you are expected to search for and digest resources to learn ANTLR on your own.

An Eclipse project containing the basis for an AST data structure has be created for you already. The root of the AST is the `Program` node. Take some time to familiarize yourself with the code provided to you in the `BrainfuckAST` project (see `https://github.com/SE421/extra-credit-assignment1/tree/master/BrainfuckAST`). In particular you will be editing the `Brainfuck.g4` file in order to generate the `BrainfuckLexer.java` and `BrainfuckParser.java` source files. Do not edit the AST data structure, since code in Problem 2 depends on the basic structure provided to you already. The `ParserSourceCorrespondence` class is used to record the character offsets of commands in the Brainf*ck source file in order to make graph selection events possible for Problem 2.

The `Brainfuck.g4` grammar file has been complete except for the Brainf*ck loop commands. For Problem 1, you must complete this file and generate the updated lexer and parser source files.

To test if you have completed this task correctly, you can use the Brainf*ck AST interpreter implementation included in the project. The class `TestInterpreter.java` contains multiple unit tests that use the Brainf*ck interpreter to test your populated AST. You should add at least one unit test that compares a Brainf*ck program that prints your name (see `https://copy.sh/brainfuck/text.html`) to the expected output.

In your report, include a link to your completed `Brainfuck.g4` file and a list of any resources your found particularly useful for learning ANTLR.

References

[1] "Brainfuck," https://en.wikipedia.org/wiki/Brainfuck, accessed: 2019-10-04.

[2] "Brainf*ck Development Tools," https://marketplace.eclipse.org/content/brainfck-development-tools, accessed: 2019-10-04.

[3] "Antlr," http://www.antlr.org, accessed: 2019-10-04.

[4] "antlr4ide," https://github.com/antlr4ide/antlr4ide, accessed: 2019-10-04.

Problem 2

(20 points)

Now that you have a working Brainf*ck parser populating an AST, you can build a program graph for a given Brainf*ck program. A skeleton project with everything except Brainf*ck loops has been implemented for you in the `com.se421.brainfuck.atlas` project. We will be extending Atlas' XCSG (`https://ensoftatlas.com/wiki/Extensible_Common_Software_Graph`) tag and attribute schema for software graphs. An XCSG schema extension for Brainf*ck has been defined in the `XCSGExtension.java` class file. Read the included JavaDoc and make sure that you understand the schema before proceeding.

First copy the generate lexer and parser source files into the `com.se421.brainfuck.atlas` project's `com.se421.brainfuck.atlas.parser` package. You will have to update the package name and re-organize imports to make the project compile successfully. The `com.se421.brainfuck.atlas` project also contains a slightly modified copy of the AST data structure.

The `ASTNode` abstract class has an added method signature of:
`public abstract Node index(EditableGraph graph, Node containerNode, SubMonitor monitor);`

The `BrainfuckIndexer` class registers a new program indexer to Atlas' indexer set which calls the `index` method on the AST's root `Program` node. Since the AST forms a tree, this method can be used to recursively walk the AST and construct the program graph. Since an Atlas indexer requires an Eclipse project nature (example a Java project or C project), we provide another Eclipse plugin project called `com.se421.brainfuck.eclipse` that registers a Brainf*ck project nature, which contains Brainf*ck program files that end in a ".bf" file extension.

Your task is to complete the index method in the `LoopInstruction` AST node. To manually test your updated indexer, right click on the `com.se421.brainfuck.atlas` project and select `Run As...` and then select `Eclipse Plugin Project`. This will launch a new Eclipse instance with your current plugin project's code installed into Eclipse. In the new runtime Eclipse instance navigate to `File` then `New` and select `Other...`. From the available categories expand `Brainfuck` and select the `Brainfuck Project`. Inside the `src` folder add a file with the extension of ".bf" and add the code of your Brainf*ck program. Next navigate to the `Atlas` menu and select `Manage Project Settings`. Enable your new Brainf*ck project for mapping and press `Save and Re-map`. If you correctly applied the XCSG.sourceCorrespondence attributes you could use the Atlas Control Flow Smart View to show the control flow graph of your program by clicking on the Brainf*ck program's source code. Alternatively, from the Atlas Shell could run:
`show(cfg(universe.nodes("XCSG.Brainfuck.ImplictFunction"))).`

Since the Brainf*ck language is extremely verbose you may choose to combine non-branching instructions into a single "basic block" (see `https://en.wikipedia.org/wiki/Basic_block`). This step is not required, but it may help with visualization and debugging. If you do choose to do this step, make sure that you apply the `XCSGExtension.Instructions` tag defined in the XCSG extension.

Finally, since the Brainf*ck language is extremely simple, the control flow graph with the XCSG extension tags applied correctly is all of the information required to interpret a Brainf*ck program. The `com.se421.brainfuck.atlas` project contains a graph based interpreter that you can use to check if your program graph was created correctly. If you can successfully execute a program from the graph you constructed (showing in Figure 1), it is very likely that your implemented is correct. The graph based interpreter starts execution given a control flow node, so select the control flow root of a Brainf*ck program and then

execute the interpreter from the Atlas Shell.

In your report, include a screenshot of the CFG produced for the Brainf*ck program that outputs your name from Problem 1 and a short description of any design decisions (such as whether or not you implemented basic blocks) you included in your indexer.

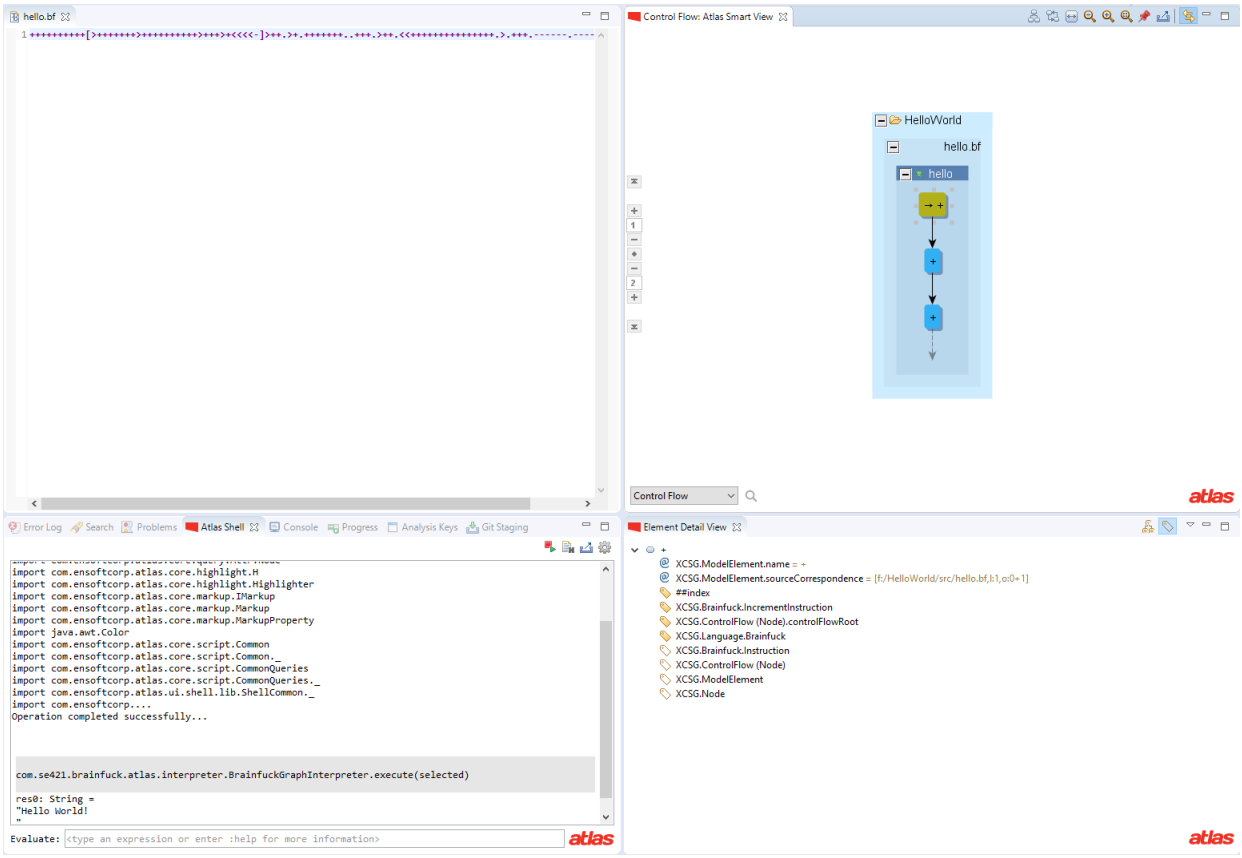

Figure 1: Graph-based Brainf*ck Interpreter

Problem 3

(5 points)

(4 points) Use your solution from assignment 3 to programmatically count the paths of the Brainf*ck in Listing 1. Include a screenshot of your path counter output result on the Atlas Shell for the program.

Note that the code shown in Listing 1 can be found at: `https://raw.githubusercontent.com/SE421/extra-credit-assignment1/master/listing1.bf`.

(1 point) The program in Listing 1 is a special type of program. What type of program is shown in Listing 1 aside from the fact that it is written in the Brainf*ck language.

Listing 1: Brainf*ck Program

```
-->+>+++>>+>+++>+>+++>>+>+++>>>+>+>+>++>+>>>>+++>+>>+++>+>+++>>+++>+++>>+>>+>++>+>++>>
>>+++>+>>>>+++>+++>>>>+>>+++>+>+++>>>+++>>++++++>>+>>+++>+>>>>+++>>+++++>>+>+++>>>++>
>++>>+>>++>+>+++>>>++>>++++++++++++++>>+>>+++>+>+++>+>+++>>>++>>++++>>+>>+++>+>>>>+
++>>+++++>>>>+++>>>>+>+>+++>>++++>+>>>>++++>+>>>>++++>+>>>>++++>>++>+++>+>+++>+>++>++>>
>>>>++>+>+++>>>>>+++>>>++>+>+++>+>+>+++>>>>>>+++>>>+>>>+++>+>>>>+++>+>>>+>>++>+>+++
++++++++++++++++>>>>+>+>>>+>>+++>+>+++>>>++>>+++++++++>>+>>+++>+>>>>+++>>++++++>>>+>
++>>+++>+>+>++>+>+++>>>>>+++>>>+>+>>++>+>+++>>>++>>+++++++++>>+>>++>+>>>>+++>>+++
+>>+>++>+++>>>>>>++>+>+++>>+>++>>>>+>+>++>+>>>>+++>>+++>>>+[[->>+<<]<+]+++++[->++++
++++<]>.[+]>>[<<+++++++[->++++++++++<]>-.----------------------->-[-<.<+>>]<[+]<+>>>
]<<<[-[-[-[>>+<++++++[->++++++<]]>++++++++++++++<]>+++<]++++++[->+++++++<]>+<<<-[
->>>++<<<]>[->>.<<]<<]
```