Assignment 3: Critters
September 28, 2018

OVERVIEW AND GOALS

The purpose of this program is to produce a simulation of “Critters,” or game objects, with a wide variety of behaviors and movements. In essence, the project works to provide a framework for the display and change of a number of user-defined objects that can clearly be differentiated through the use of preset commands. Programatically, this project works to teach the implementation of File IO when reading from user text files, as well as introduce the concept of an Interpreter. This project shows how interpretation can lead to abstraction, and eventually the creation of a new programming “language” altogether (what was created in the final result of the implementation of the Interpreter class). Finally, the critter outputs that are simulated in the field work to both amuse and entertain the user.

We had several personal goals throughout the course of this project. Firstly, we wanted learn more about game design, game simulation, and game strategy – a popular computer simulation this project reminded us of was Conway’s Game of Life. In addition, we wanted to use this project as an opportunity to learn more about the field of interpreters and compilers; after all, the implementation we were tasked with is simply a way of abstracting complicated code into a simpler form, the basis of language design. Finally, we wanted to showcase our abilities to test robust code with the inclusion of our test harness.

SOLUTION DESIGN

High Level Description

Our interpretation of this project was implemented in four key steps –

1. Reading the .cri files within the loadSpecies method of the Interpreter and storing the commands in an object to be returned
2. Interpreting command verbs within the executeCritter method (ex: hop)
3. Parsing formatted numbers (ex: 5 vs +5) and executing commands based on these parameters
4. Designing a multifaceted test harness with various edge test cases, as well as implementing given interfaces to control input and output (ex: Critter interface)
For the reading of the .cri file, the BufferedReader class from the java.io library was used for
efficient read times using streams. In addition, substrings and regular expressions were also used
when parsing strings, and the JUnit library was used for testing different parts and functions of
the interpreter itself.

**Implementation Design**

Implementing the loadSpecies method was fairly simple with the BufferedReader class, as we
simply traverse through each line of the file and add relevant ones to an attribute ArrayList that is
returned. The executeCritter method, however, is not as straightforward.

To start, we loop through each command, starting with c.getNextCommand() - 1 (subtracting one
to account for indexes starting at 0) to the end of the command list. From there, we implement
every method that relinquishes its turn when it is called. For the hop and directional methods, a
straightforward call to the respective abstracted method sufficed. For methods that took in
additional parameters, however, we first split each command into an String array with the
command verb (ex: infect), and the expected parameters. For example, “ifenemy 0 5” would be
split into [“ifenemy”, “0”, “5”]. Next, the n value is parsed according to the first character (if the
first character is “r” a register is called, + or - are relative jumps, and a simple number is an
absolute jump). Finally, we check if the required conditions to execute the method are met (for
example, an enemy must be in front of a critter for the infect method to be invoked), and a critter
function is invoked if all requirements are passed. Finally, the nextCodeLine is incremented and
the method returns so as to relinquish its turn. For methods that do not relinquish their turn,
however, a return statement cannot be called. Instead, i (the for loop incrementer) is set to
c.getNextCodeLine() - 2. It is necessary to subtract two because of the for loop index starting at 0
instead of 1, and the fact that the loop increments at every pass. Therefore, the code will resume
at the new specified code line without breaking the loop or relinquishing the turn. In terms of
actually checking the conditions of each method, most only required a simple if statement – for
example, checking the “ifeq” command simply required making sure that the content of the two
specified registers were equal to one another.

**Critter Design**

In order for us to create our own critter, we had to prepare for the most possible contingencies.
Firstly, we ensured that the critter had the ability to eat – without this, our critter would soon
perish due to the decreasing amount of turns it would have as its state turned to hungry and
starving. We created this flowchart to organize how we wanted our critter to act:
While admittedly basic and not exactly in the format of a formal flowchart, we believe this simple design can capture many of the states of action the critter will be in. Even though implementing the Critter programming language is complicated, the objective of the game is rather simple: survive. In order to do so, the critter focuses on resolving issues of hunger first as those cause the biggest limits on the critter. After that, it seeks out other critters to infect. The critter is designed to be primarily offensive, playing a little recklessly and without much thought for defensive maneuvers. In the original iteration, we used relative line jumps to check each bearing for an enemy to eat, but we realized quickly this wouldn’t be practical as not all the commands were executing correctly in the order we wanted to. So we utilized the go command to organize the critter code better. After testing it, the Ajjarapu critter does behave as intended. It is very aggressive, quickly consuming all other critters. However, it didn’t function well against Rover because it wasn’t infecting. Thus, I modified the code to infect other critters more aggressively. Something we could work on in the future is adding a little more defensive strategy to the critter to increase its survivability.
RESULTS

Assumptions
When reading .cri files in the loadSpecies method, it is assumed that the contents are formatted as specified in the manual and all commands are legal, along with n values. For example, if the command “ifenemy” is called, then it is assumed that b (bearing) and n (line value) are also passed as parameters. In addition, it is assumed that all commands are ordered in a legal way that does not set the next code line to be out of bounds. Because the left and right methods require no additional functionality besides being called by the interpreter, no unit testing was done on them and it is assumed that they (as well as all abstracted Critter functions) work correctly (black box testing was used to check these methods). It is also assumed that the program will be used in a setting with a valid simulator, as the interpreter would perform differently otherwise.

Problems Encountered
A central problem encountered in this project that took a significant amount of time to circumvent was fully understanding the given API and what our tasks actually entailed. Due to the high level of abstraction that was given to us by the simulator, it was not clear whether we had to actually implement commands (such as hop), or simply route them to an instance of the Critter Interface that was created by the simulator. In addition, we had significant issues with commands such as “go” that jumped to the nth instruction, because the code that looped through our commands started at 0 instead of the next code line. Therefore, the interpreter either skipped lines, didn’t jump to the next line and restarted interpretation every turn, or simply threw errors before we realized what we were doing wrong. Finally, a big problem encountered was actually wrapping our heads around how to test and simulate the Critters and world outside of black box testing. Because so many functions of Critter objects (such as changing the x and y coordinates to create desirable spawn positions) were abstracted away, designing a harness that thoroughly tested the Interpreter took a significant amount of time to think about. Because neither of us had done unit testing previously, learning JUnit and writing our code in a way that could be broken into pieces also posed issues that we had to deal with.

Scope and Limitations
A limitation we have with our interpreter is that errors will be thrown when there are incorrect commands, and when commands are placed incorrectly. For example, a critter with only one command, “go -1”, would throw an error as the nextCodeLine would be invalid (0), and another error would be thrown if the command “go” was called without the required n value. If a call to go
to a line in specified in a register was called before a register was written to, an error would be posed as well due to the default instantiation of registers to 0. In addition, the loadSpecies method is also limited by the max size of a String in Java, as the file contents are stored in a string. Finally, the abstraction of the interpreter could have been slightly improved, as the code to evaluate the different syntaxes that n could have was shared between multiple methods that probably could have been captured in a singular method. However, the scope of this interpreter is actually reasonably large – as long as it is paired with a valid simulator and the critter file is within a reasonable size, the interpreter should correctly read and execute legal commands for any given critter. In addition, the code is highly readable and both the interpreter and tests are easy to add upon – in essence, it is scalable within reason.

TESTING

In terms of testing, a good starting point was obviously black box testing with controlled inputs. To do this, we made critters with simple commands and visually looked to see what happened when the code was run. For example, in the first version of our Interpreter, a critter with only the commands “hop” and “left” would only turn left every turn, instead of hopping one turn and rotating the next. Therefore visually, we knew something was wrong with our interpreter. We implemented white box testing as well by using print statements to ensure every command being called was sequential, but we soon realized a more robust implementation was needed. To do so, we decided to emulate the simulator as a kind of “test harness” and use JUnit tests to individually test each command that the interpreter handled. Firstly, we made a “TestCritter” class that implemented the Critter interface, and simulated the implemented methods so as to test the interpreter. For example, the “getCellContent” method returns an enemy when checking for the front cell of the critter, and returns an ally at the rear. In addition, this TestCritter is able to handle passed commands when it is instantiated, and also simulates things like registers and manipulating the code lines, so that the interpreter can be tested on the critter. We implemented JUnit tests that went through each command and tested both regular and edge cases to make sure that the interpreter worked on our “simulated” TestCritter. For example, let’s assume that the TestCritter object was instantiated with the command “ifhungry 5”. Because we implemented the getHungerLevel method in TestCritter to always return a state of starving, we can test the interpreter by using the executeCritter method on TestCritter, and asserting that the nextCodeLine of the TestCritter was set to 5. In this fashion, each method was tested with all formats of n in addition to registers. Finally, the loadSpecies method and manipulation of registers were also tested by using assert statements to compare expected and actual values, as well as making an actual .cri file that was read with the loadSpecies method.

A limitation that our JUnit testing has is that the infect/eat methods and the hop method cannot be tested at the same time. Because the getCellContent method of our TestCritter had to be simulated manually, the square in front of the critter could not return empty for hop and have an
enemy to test the infect/eat methods at the same time. Therefore, we decided to simulate an enemy for the square in front of the TestCritter, and have the JUnit test for hop test for if the critter did not hop in this case. Another weakness in our testing was the fact that it was hard to simulate two critters interacting without using simple black box testing – perhaps if we had written a full simulator, we could have controlled spawn points to fully test interactions in a non isolated fashion.

**Interesting Results**

When researching methods to parse commands that included n in its different formats, we discovered the concept of regular expressions. We learned that with the simple command “\d”, for example, we could in fact encapsulate all digits in our search instead of hardcoding each individual digit to be tested. In addition, while making our critter, we discovered that we could mimic and simulated much more complicated movements than we had originally anticipated. In essence, this highlighted the fact that despite the popularity of abstraction, every program made in an abstracted language can also be recreated in a lower level language. We discovered that the infect and eat commands were especially powerful, but it is ambiguous who wins if two critters facing the same direction eat or infect each other at the same time – it would be interesting to see if there was a way this could be handled by our implementation of the simulator.

**Pair Programming**

- 9/16 – Ashwin drove for 0.5 hours, Nikhil drove for 0.5 hours
- 9/19 – Ashwin drove for 0.75 hours, Nikhil drove for 0.25 hours
- 9/22 – Ashwin drove for 0.5 hours, Nikhil drove for 0.5 hours
- 9/24 – Ashwin drove for 0.1 hours, y drove for 0.9 hours
- 9/25 – Nikhil spent 1 hour working individually
- 9/26 – Ashwin drove for 1 hour, Nikhil drove for 0.5 hours
- 9/29 – Ashwin spent 2 hours working individually

Overall, both of us have enjoyed the experience of pair programming. Because of the driver being constantly overlooked, debugging and catching simple errors was much easier than it would be programming alone. In addition, the creation of new and original ideas was much easier than with a singular mind, but also tended to make implementation a tad slower due to thorough discussion about what was to actually be done. Overall, it resulted in more quality and refined code with the tradeoff of slightly greater development time. It was slightly harder due to the fact that one of us was absent for a week, but Skype and version control greatly mitigated the issue.

**CITATIONS**
No additional sources were used in the implementation of this project