Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Assignment 3 - Critter Report

Introduction/Goals

This project was different from the previous couple of projects in that it was completed in teams of 2. Our program acts as a compiler because it takes in code in the Critter language, checks to make sure there are no errors, and runs the code. There were several requirements for this project. First, an interpreter was created to be able to read a set of simple instructions and perform the appropriate behavior for whichever critter was called. Second, a test harness was created to ensure that our interpreter worked correctly. Finally, a critter was designed and created in order to compete with other critter submissions. Our team goals for this project included learning about how compilers work and creating our own critter that would be successful at CritterFest.

Solution Design

The solution design includes 11 different methods within an `Interpreter` class. This class implements an interface named `CritterInterpreter`. The reason our design was so modular was because it would be easier to test small methods separately. As we were coding, it was easier to discover flaws in functionality in the new code and it was very easy to test every method after the code was implemented. The 11 methods are listed as follows:

- `processParameter(String param, Critter c)`
- `initializeConditionals()`
- `initializeMutators()`
- `initializeActions()`
- `initializeNumParameters()`
- `executeCritter(Critter c)`
- `getBooleanValue(String method, Critter c)`
- `getBooleanValue(String method, String param, Critter c)`
- `getBooleanValue(String method, String param1, String param2, Critter c)`

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

- *loadSpecies(String filename)*
- *detectErrors(ArrayList<String> code)*

Next, we discuss each method's purpose.

processParameter(String param, Critter c)

The `processParameter()` converts parameters taken as `Strings` into `ints` in two ways. First, it checks whether the first character of the `String param` is *"-"* or *"+"*. If so, the parameter is processed and returns new code line relative to the current code line. For example, if the parameter was the `String` *"+2"*, the method returns the code line that is 2 lines ahead of the current code line. Second, it checks whether the first character of the `String param` is *"r"*. If so, the parameter is processed as the value stored in that register.

Initialize Methods Using ArrayList

This section includes 3 methods that initialize `ArrayLists`: `initalizeConditions(), initalizeMutators(), initializeActions()`. The `initalizeConditions()` method returns an `ArrayList` containing all the conditional methods (beginning with if). The `initalizeMutators()` method does the same thing for the mutator methods, and so does the `initializeActions()` for the action methods. The purpose of these 3 initailize methods is to classify the similar commands so we can compare the commands in the code line to each of these `ArrayLists` so we know what to do with it.

initializeNumParameters()

This creates a `HashMap` that maps the list of callable methods in the `Critter` language with the amount of variables they are expected to take. This method makes 4 `HashSets<String>` and maps them to 0, 1, 2, and 3 within the `HashMap<Integer, HashSet<String>> numParameters`. In this `HashMap`, each `Set`'s `key` corresponds with the amount of parameters that each command in the `Set` takes. For instance, the commands `hop` and `left` would be in the `Set` mapped to `key` 0 because they do not take in any parameters, and the

commands `ifangle` and `ifeq` are in the `Set` mapped to `key` 3 because they take in 3 parameters.

executeCritter(Critter c)

This method retrieves the `Critter's` behavior code, executes it, and calls one of the action methods before it returns. It checks whether the first word of the first line is an action method or not. If not, then it checks if it's a conditional method or a mutator method. If it is conditional, then we check how many parameters the line has. The overloaded `getBooleanValue()` method with the correct number of parameters is then called. If it is a mutator, then we change the values of the registers by either writing to it, adding two register values, subtracting two register values, incrementing a register value, or decrementing a register value. We repeat this process for the first word of the next line until we reach an action method unless we are told to go to a specific line n. However, if the first word of the first line is an action method, then we execute the action method and return.

getBooleanValue(String method, Critter c)

This method implements the conditional methods of the `Critter class` when the method does not have any parameters. The only conditional method that does not have any parameters is `ifrandom`. So we compare the `String method` to `ifrandom` and if they are equal, we return the `ifRandom()` method applied on the `Critter c`. If they are not equal, an error statement is printed letting the user know which command rendered the error.

getBooleanValue(String method, String param, Critter c)

This method implements the conditional methods of the `Critter class` when the method has one parameter. The conditional methods that have one parameter are `ifempty`, `ifally`, `ifenemy`, and `ifwall`. It first checks whether the parameter is an valid integer or register. Next it checks if the `String method` equals one of these 4 methods and returns the boolean value of whether the method is true or false for the corresponding parameter. If the `String method` does not equal one of these methods, an error statement is printed letting the user know which command rendered the error.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

getBooleanValue(String method, String param1, String param2, Critter c)

This method implements the conditional methods of the `Critter class` when the method has two parameters. The conditional methods that have two parameters are `ifangle`, `iflt`, `ifeq`, and `ifgt`. It first checks if both parameters are either valid integers or registers. Next it checks if the `String method` equals one of these 4 methods and returns the boolean value of whether the method is true or false for the corresponding parameter. If the `String method` does not equal one of these methods, an error statement is printed letting the user know which command rendered the error.

loadSpecies(String filename)

This method takes in a file, reads it, and returns a new `CritterSpecies` object. It does so by using a `Scanner` object to read an input file. If the file cannot be found the error statement `"File Not Found."` is printed. If the file has no next line, then the error statement `"Empy File."` is printed. We read the input file line by line and store it in an `ArrayList` of `Strings`. A correct input file would have the critter's name on the first line, its commands on the next few lines, and have a blank line followed with a description. The description is useless for the program, so we take care of this by breaking once and empty line is found. Thus, the description is not read. It finally returns a `CritterSpecies` object with its name and an `ArrayList` called `code` consisting of the commands.

detectErrors(ArrayList<String> code)

This method parses through the critter code to check if it will operate without bugs. There are 7 for loops integrated within this method. The first for loop checks if the first word in each code line is a valid command. It does so by checking if the word is `"go"` or if it is contained in either the conditionals `ArrayList`, the mutators `ArrayList`, or the actions `ArrayList`. If it is not a valid command, a helpful error message is printed. The second for loop checks if the remaining words in the current code line are valid registers (`r1-r10`). It also asserts that if the first word is not go, then the following parameter(s) may not be preceded by a `"+"` or `"-"`. In both cases, a relevant error message is printed if one of these things goes wrong. The third for loop goes through the `HashMap` called `numParameters` and checks to see if the number of parameters following

the command is correct. If not, an error message is printed letting the user know the command does not expect that many parameters. The fourth for loop makes sure that the conditional methods that take in one bearing parameter have a valid bearing (0, 45, 90, 135, 180, 225, 270, 315). If not, an error message is printed letting the user know the command does not have a valid bearing. The fifth for loop is similar to the previous for loop except it takes two bearing parameters and checks if both are valid bearings. The sixth for loop makes sure the parameter of `write()` is a valid register. Similar to all the previous loops, it prints an error message if a register is invalid. The last for loop is like the previous for loop except it is designed for the rest of the mutator methods because they take in two registers as parameters. It prints an error message if it encounters an invalid register.

Scope and Quality of Solution

Assumptions

The solution made a few assumptions regarding the inputs the user used. The file had to be of type "`.cri`". The solution also assumed that the user had some type of set of instructions on how to write the "`.cri`" file. For example, our code let the user know that a bearing value of 60 was invalid, but it did not list all the valid bearings. Here, the user would have to know that the only valid bearings were 0, 45, 90, 135, 180, 225, 270, and 315. The same issue holds for register values. We simply let the user know the register was invalid, but we did not list valid register values. It is reasonable to assume the user had this type of information beforehand, but just made a silly mistake while typing up the input.

Scope

The solution took in any combination of commands as long as the first line was the `CritterSpecies` name and each command had valid parameters. As a team, we experimented with many different types of Critters for fun and for the purpose of creating a Critter that would have a very good chance of surviving while going up against other Critters.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Quality

The solution produced seemed to be very efficient. Appropriate data structures were used. For example, `ArrayLists` were used to store simple lists of `Strings`. This was a good data type to use, because it stored `Strings` in order, and it was easy to access and manipulate the given values. But when we had lists that had enumerable properties, we used a `HashMap`. We mapped an integer (the number of parameters) to its corresponding methods listed in a `HashSet`. All the methods with 0 parameters were listed in one `HashSet`, the methods with 1 parameter were listed in another `HashSet`, and so on. This allowed us to have a very organized `Map` of `Sets` that corresponded with relevant values.

Software Test Methodology

Black Box Testing

For this program, black box testing was not the most effective method of testing. However, we could get an idea of how each Critter moved and if it executed the correct commands using the simulator. The simulator was a great tool to have, but it was not very useful for testing because of the randomness created when placing the Critters in the simulated world.

White Box Testing

White box testing was generally used to test specific components (similar to unit testing) of the code to check whether cases where the code seemed dubious were correct. Even though many of these cases were required as per the assignment, they can all be tested. All our tests were located in one test class called `InterpreterTest`. We also implemented the `Critter` interface into a class called `TestCritter`. This `Critter` object had extra methods that made it easy to observe and manipulate values, so that it would operate very predictably during tests. Our tests could be split into either testing the `loadSpecies()` method or the `executeCritter()` method. Below is a table listing the segment of code that was tested, the input, the actual output, and if it matched with the expected output.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Table of `loadSpecies()` Tests:

Code Segment Tested	Input File	Output
Tests if `loadSpecies()` method reads the input file correctly	TestCritter1.cri	The input file was correctly printed.
Tests if `loadSpecies()` method reads the input file correctly with edge cases like registers, relative jumps, ifangle, and infect with multiple numbers of parameters.	TestCritter2.cri	The input file was correctly printed.
Checks if error statement is printed when an invalid command is input.	InvalidCommandTest.cri	Error message "`Command not found: eaat`" was printed as expected.
Checks if error statement is printed when parameters are words.	InvalidParameterTest1.cri	Error message "`Invalid Parameter: five`" was printed as expected.
Checks if error statement is printed when parameters are not integers.	InvalidParameterTest2.cri	Error message "`Invalid Parameter: 5.5`" was printed as expected.
Checks if error statement is printed when parameters are invalid registers.	InvalidParameterTest3.cri	Error message "`Invalid Parameter: r11`" was printed as expected.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Checks if error statement is printed when parameters are registers being used for relative jumps.	InvalidParameterTest4.cri	Error message "`Invalid Parameter: -r1`" was printed as expected.
Checks if error statement is printed when conditional methods with one bearing parameter use an invalid bearing.	ValidBearingTest1.cri	Error message "`Parameter -12 not expected for command ifally`" was printed as expected.
Checks if error statement is printed when conditional methods with two bearing parameters use an invalid bearing in the first parameter.	ValidBearingTest2.cri	Error message "`Parameter 360 is not a valid bearing for ifangle`" was printed as expected.
Checks if error statement is printed when conditional methods with two bearing parameters use an invalid bearing in the second parameter.	ValidBearingTest3.cri	Error message "`Parameter 44 is not a valid bearing for ifangle`" was printed as expected.
Checks if error statement is printed when mutator methods with one register parameter use an invalid register.	ValidRegisterTest1.cri	Error message "`Parameter 45 is not a valid register for write`" was printed as expected.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Checks if error statement is printed when mutator methods with two register parameters use an invalid register for the first register.	ValidRegisterTest2.cri	Error message "`Parameter 30 is not a valid register for add`" was printed as expected.
Checks if error statement is printed when mutator methods with two register parameters use an invalid register for the second register.	ValidRegisterTest3.cri	Error message "`Parameter 50 is not a valid register for add`" was printed as expected.
Checks if error statement is printed when a parameter is preceded by a "+" or "–" in a method besides go.	ParameterPrefixTest.cri	Error message "`Parameter: +5 not expected for command ifally`" was printed as expected.

Rohit Bhatta
Matthew Prost
9/25/15
CS 314H

Table of `executeCritter()` Tests:

Code Segment Tested	Input File	Output
This shows that the negative relative jumps work as intended.	TestCritter3.cri	The critter hops, rights, lefts, then repeats the left command as expected.
This shows that the positive relative jumps work as intended.	TestCritter4.cri	The critter hops, skips line 3, lefts, then repeats as expected.
This shows that the register values are properly processed and accessed.	TestCritter5.cri	The critter writes the value 2 to r3, rights, lefts, infects, then repeats all commands after line 1, as expected.
This shows that the conditional methods with two parameters work correctly.	TestCritter6.cri	The critter eats, infects, lefts, and skips line 5 as expected.
This shows that the conditional methods with three parameters work correctly.	TestCritter7.cri	The critter ends up with two equal parameters r1 and r2, then infects, lefts, and skips line 7 as expected.
This shows that the conditional methods with three parameters work correctly.	TestCritter8.cri	The critter writes register 1 to 0 and increments by 1 until it is equal to register 2, then it decrements twice as expected.

All these tests called either the `loadSpecies()` or `executeCritter()` method on an object of type Interpreter and checked whether the actual output matched the expected. Testing the `loadSpecies()` method was tedious but relatively simple because the `detectErrors()` method in the `Interpreter class` did most of the actual work. Testing merely made sure invalid commands and parameters were recognized and printed as errors. There were several cases where bugs were found because our output did not match what we expected. Eventually, all discovered bugs were fixed and the program seemed to be rid of errors. White box testing was especially effective thanks to the original solution design being modular.

Critter

The critter hops in a straight line until it ends up next to an enemy. If the enemy is in front, then it infects it, if it is not, and the enemy is facing it, it turns towards the enemy, so that it has less of a chance of infecting an ally after it is infected. This makes it take a while to kill the last few critters on the map because it has a tendency to avoid the enemies if possible.

Conclusion

This assignment revealed many insights about how to effectively approach a programming project. Careful planning can be very useful because it minimizes large extraneous errors and backtracking. At the beginning of the project, we jumped right in, but we had to change much of our initial code because of misimplementation. Later in the project, we found that by planning ahead, we could smoothly implement new code with very few new errors. We also found how useful testing can be. Our test harness found many more bugs than we were expecting, showing how valuable thorough testing is to creating a high quality finished product. We also found that sharing the load between two people during pair program is tricky but valuable. In order to find the optimal results the load should be shared evenly, and it should be shared regularly. When we first began pair programming, we did not switch programmers at regular intervals and always waited a long time between switching. By switching programmers at intervals of about 10 minutes, both partners could remain engaged, and their overall understanding and vigilance increased. These insights can be implemented in future projects to help provide a high quality solution in the smallest amount of time possible.

Pair Programming Log

Date	Start Time	End time	Accomplishments	Notes
9/14/2015	1:00 PM	2:00 PM	We found overlaps within our schedules, established weekly objectives, and learned about each others' relative programming abilities.	Matt has more technical knowledge than Rohit, but Rohit demonstrates an ability to run good tests on programs.
9/16/2015	3:00 PM	5:00 PM	We began on the loadSpecies and executeCritter methods.	We need to work on preplanning and switching programmers at 10 minute intervals.
9/18/2015	3:30 PM	5:30 PM	We finished loadSpecies and planned for the implementation of the executeCritter method. We also discussed our plans for a test harness.	We plan to make the code very modular, so we will be able to test each of the individual pieces.
9/21/2015	4:00 PM	5:00 PM	We worked on executeCritter. Due to a few misconceptions and oversights, we had to change some of our existing code.	Major deficiencies in our programming time so far have been our lack of planning, misunderstanding of instructions, and fixing errors that could have been avoided.
9/22/2015	11:00 AM	1:00 PM	We worked on executeCritter.	We went over major topics today that would make programming easier. These included Sets, Maps, and Regular Expressions.
9/23/2015	3:00 PM	6:30 PM	We finished executeCritter and began work on the test harness.	
9/23/2015	10:30 PM	1:30 AM	We tested loadSpecies and fixed some bugs.	We found the test harness to be even more useful than expected. It found many more bugs than we anticipated.
9/24/2015	10:30 AM	12:00 PM	We tested executeCritter and began our report.	We switch programmers every 10 minutes, but we write different sections of the report simultaneously.
9/24/2015	12:30 PM	1:30 PM	We created the TestCritter class and worked on the report.	
9/24/2015	3:30 PM	5:30 PM	We worked on the report and tested executeCritter using our TestCritter class.	
9/24/2015	1:30 PM	4:00 PM	We worked on the report and finished our CritterFest Critter.	