Project 3 Report

Atharva Pendse & Zhaosong Zhu | CS 314H

Assignment Goals

In the Critters project, the objectives are to implement an Interpreter class, design a test harness to test the functionalities of Interpreter and to design a critter that will later be used for the competition. This project introduces us to the idea of developing a test harness, which is a relatively new topic for both of us. Our personal goals include getting familiar with project partners, designing efficient algorithms for Critters to survive and fight, learning to use APIs, and finishing bonus Karma activities to gain more knowledge about the entire project.

Solution Design

The design of Critter project involves three aspects: *Interpreter* class, which includes *loadSpecies* and *executeCritter* method, critter commands, and test harness design. The *Interpreter* class aims to process imported .cri files and execute certain lines of commands based on the critter design. The critter .cri files contain a set of commands provided by assignment instructions, which we use to create effective battle critters. To see if the various instructions are executed correctly, we also develop a test harness including a redesigned Critter and CritterSpecies class

A. High-level details on implemented classes and methods

1. ***loadSpecies(String filename)***

   The loadSpecies method reads the .cri text file and stores each new command as a String in an ArrayList. We store each of the commands as a separate string, adding them to an ArrayList.

   However, since the .cri file always saves the first line in each critter file as the name of that critter species, we attribute the first line of code to String species. Then, before adding the actual commands line by line into the ArrayList, we add an empty string “” for index 0.

   The objective is to make it easier to invoke methods such as “go n”, which is equal to the array index number, reducing errors due to +1 or -1.

2. ***getNextCodeLine() and setNextCodeLine(n)***

   Since each time a return statement is needed to execute some commands, including hop, left, right, infect, eat, it is not possible to merely use an integer to keep track on the line number of commands. Therefore, the getNextCodeLine() and setNextCodeLine(n) methods are used to trace lines.

For all the instructions listed above, *setNextCodeLine(i+1)* will perform on the Critter object after the corresponding command such as left, right, eat, is executed and before the return statement. In doing so, the # of line which should be executed next is set, so that when the *executeCritter* method is invoked again, the commands *int i = c.getNextCodeLine();* and *String cur = commands.get(i);* will be able to obtain the correct code to run even when reentering the method. The old int i will not exist anymore because once the method is exited, all data will be deleted.

The objective of this design is to allow the interpreter to keep tracking line number even when the *executeCritter* method is exited and invoked again.

3. Logic within critter file
   Inside the .cri file for our critter, *ifenemy* and *ifangle* methods are often used to determine whether there is an enemy at a certain location and whether they are facing at us. Starting from line 2 to, *ifenemy b n* instruction is executed to determine whether there is an enemy at different bearings, including all possibilities from bearing 0° to 315°. For all cases except 0°, if the *ifenemy* command is evaluated to true, then it will jump to an *ifangle* command to see if the enemy is facing the critter itself. For example,

   ifenemy 45 +2
   go +4
   ifangle 45 225 +2
   go +2
   go 75
   ifenemy 90 +2

   Under these three lines of commands, if there is an enemy at 45° bearing of the adjacent square, the code will try to see if the relative bearing of the two critters is 225°, which means the enemy is facing us and it might attack us at right next round. If the *ifangle* is evaluated to true also, it will go to line 75, starting from which there is the subcode we designed to determine what to do. If not, then the code will go to *go +2* and thus jump to evaluate another angle.

   The subcode, which is a section of code that is frequently called on in many situations, ranges from line 76-87 within the *Test.cri* file. If we find an enemy critter facing our critter, we can use the subcode to find if there is an ally, wall, or empty right in front of the heading of the critter. If it's an ally or a wall, then we turn right and jump back to line one, allowing some other actions to be taken based on the code design. Otherwise, if it's empty, then the critter will hop ahead to avoid being eaten/ infected.

   The objective is to detect the enemy at all possible locations around us, and if an enemy exists, some repeated algorithm can be used to prevent infecting.

4. Handling Exceptions/ Edge cases

We created two methods- gotostepn and parseint-in the Interpreter class, which we used for detecting number format exceptions. If the string that is provided as an argument to these methods isn't an integer or an invalid string, the method will throw a numberFormatException.

In case the provided behavior file is empty, the loadCritter method returns null. In case the input file is invalid or non-existent, the method throws an IOException.

B. Abstractions

1. **Substring methods**

   The substring method is used pretty often throughout the executeCritter method. It mainly serves two functions.

   The first function is that since we already know what commands are available, it is useful to crop out the former part of the line of code to match with different implementations. For example, if a *go 1* is passed in, then a *substring(0,2).equals("go")* can be used to direct to implementations for the go command.

   For the other thing, substring method is also used along with parseInt. Using the same example above, substring(3) can be used to read which line it should go to, assuming everything after index 3 is a valid number. (We check that with a number format exception)

2. **The *gotostepn* method**

   We found the gotostepn method to be a valuable tool for code reuse. The *gotostepn* method takes a String *step n* and integer variable *i* as its input. It parses the String stepn for any + or - signs at the beginning of the line (for example +5 or -3) and separates the sign and the number. It then sets the new value of integer *i* accordingly and returns that value. For example, if stepn is "+9" and i is 3, the method will set *i* as 12 and return *i*. If stepn is "-3" and i is 6, it will set *i* as 3 and return i. If the first character of stepn is not a '+' or '-' character, then it will simply make i equal to the number represented by string stepn and return i.

   ifwall 0 +2
   go+2
   right
   go 1

   For example, the lines shown above helps achieve both functionalities. The *ifwall* statement is an if statement, such that if *ifwall 0* is true, then it goes to two lines after, which is *right*. If this is false, which equals to else, then *go +2* is executed, which goes to *go 1*, thus looping the codes to go back to line 1 and restart evaluating all situations.

   The *gotostep n* method saves a lot of code especially in the ifally, ifenemy, ifempty, ifwall and ifrandom commands.

Project Discussions

A. Critter Designs, Assumptions, Strength, and Weakness

As we are trying to design the critter, our idea and assumption are that for a critter to live long enough, it must be able to detect if there's any enemy exist on all bearing, ranging from 0 to 315. Then, if there is an enemy right in front of my critter, the critter will infect it. Under situations where the bearing goes from 45 to 315, we not only determine if there's an enemy but also if the enemy is facing us. This is accomplished through the *ifangle* method. For example, the code below shows that when detecting if there's an enemy at 45-degree bearing, we try to see if the enemy is facing us by using *ifangle.* If the enemy is facing us, then the relative bearing between two critters will be 225 degree. If such situation is false, then it means that the critter won't be able to infect us in one round.

```
ifenemy 45 +2
go +4
ifangle 45 225 +2
go +2
go 75
```

For the most part, after evaluating the two situations, if both of them are true, then it will direct to line 75, starting from which are the subcodes shown below. The subcodes try to cover three situations: if there's an ally, a wall, or nothing right in front of it. If there's nothing, then it will hop ahead. Otherwise, it will just make a turn, which might help critters hop to somewhere else after re-running codes from line 1.

```
ifally 0 +2
go +3
right
go 1
ifempty 0 +2
go +3
hop
go 1
ifwall 0 +2
go +2
left
go 1
```

After these implementations, our critter can win 100% under a situation where 30 enemies vs. 30 critters, regardless who goes first. However, as the enemy goes to 2-2.5x of # of critters, we find that our critters don't win as often. Interesting enough, they begin to show up pretty often near edges, which makes them vulnerable.

Another three situations are considered after this: *ifwall 0 +2, ifwall 45 +2, ifwall 315 +2.* In doing so, we try to address three situations: if a wall in front of critter, if a wall at right 45 degrees, if a wall at left 45 degrees. We believe these are the most common situations when a critter is trying to hop forward. Just in case there are situations where the critter just got stuck at somewhere there are walls on more than two sides, we add in a random turn left or right command, hopefully helping the critter to be able to get out and hop again. These codes below allows the critter to turn left or right at a random but equal opportunity.

```
ifrandom +2
go +3
left
go 1
right
go 1
```

To test the strength of our critter, we put 30 critters vs. 300 Rovers. The Critters won 4 out of 5 times if Rover moves first. If Critters move first, the critters will win every time.

However, a relative weakness of our critter is that if both the critters and enemy are of the high amount, it is unlikely that our critters will win. We believe the reasons are twofold. For the one thing, with a lot of critters on the field, not many spaces are available for the critters to move around and infect other critters. The less space, the less the mobility. For the other thing, although we do add in some considerations about wall situations, they are not sophisticated enough to just stay near the walls and infect others. Admittedly, the best design for this particular case may be just having critters that turn around and infect right at its location, which is not a general solution with such considerations.

B. Edge cases and Potential Solutions

1. **Infinite loops**

   Infinite loops occur when commands besides hop, left, right, infect, eat are repeatedly executed, thus causing the entire simulator to stop working. Since the commands that may cause the infinite loop does not invoke a return command, we can use a separate variable to detect and solve the problem within the *Interpreter*.

   Based on the fact that a return statement is not executed for a long time in case of an infinite loop, an integer p is created at the beginning of the *executeCritter* method to count the number of occasions all other methods are executed. Within the while loop, which is the condition that the line tractor variable i is smaller than the ArrayList of command, with a size of command lines +1, an if statement is added in the front. If we find the same line is executed for more than 1000 times, then we will print out an error message and thus set the command line for this specific critter to *commands.size()+1*.

   Besides, in case a situation where the line number of the critter is already out of its max line number, another return statement is added outside the while loop that tests if the line

number if larger than the max line number. Doing so will keep the critter there with no command executed until being infected/eaten.

2. **Empty command set/ file**
   Another possible edge case is that the user might submit a .cri file that is either empty or does not contain command lines. This situation is handled within the *loadSpecies* method. A try-catch structure is used to catch any Exceptions that may terminate the program. For the one thing, if the .cri file is completely empty, then the FileReader won't be able to read from any line and thus throw an error, caught by the catch part and return a *null.* For the other thing, if the critter file only has one line before a blank line, then the very first line will be read as *species,* and an ArrayList of size one will result, due to the filler "" in it. In this case, if *(commands.size()>1)* is used to check if there's at least one line of commands. If not, then the method will as well return *null*.

3. **Weird characters + lack of parameter**
   Another edge case will be incorrect command lines. The first kind of situation is when anything besides number passed in. To deal with this situation, three methods - *gotostepn(String stepn,int i), parseInt(String parse), subString(String a,int index)* - are created to catch any error that might happen when using *parseInt* or *substring* methods.

   1) ***gotostepn(String stepn,int i)***
      This method is used in situations where a command line contains a line number to jump to. For example, *go hello* is an incorrect command because it didn't specify the numerical line to jump to. In this case, an error will occur with the *parseInt* method, which will be caught and thus return a -1 as the line number. Since the i variable receives the line number, and the while loop only runs when command.size()>i>=1, the loop will stop, and the code line is set to outside possible range.
   2) ***parseInt(String parse)***
      This method is primarily used with register operations. It performs a very similar function as specified above, which is to return a -1 when error caught. So, before a register can be changed, we will test to see if the # of the register is between. If it's negative one, then we will set code line to outside possible range, which stops the critter.
   3) ***subString(String a,int index)***
      This method is used in situations where needs to read all the last part of a command line. The read result is expected to be a line to jump to or a number to be stored. For example, *ifgt r1 r2* is invalid because it doesn't specify which line to jump to. In this case, since there will be an out of bound error when trying to obtain the last part by using substring method, this recreated method is to catch the error and return "n/a" back, which will once again, be handled by the two other methods above when changing to a number.

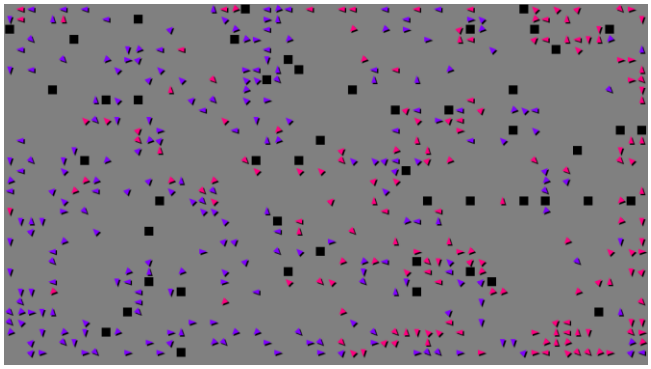
   Some minor edge cases are listed below:

1. If the go instruction or any of the ‘if’ instructions go to a line number that is out of bounds.
2. If the bearing provided as an argument is not a valid bearing (that is if it is greater than 45), or if it is greater than 360.
3. If any of the arguments provided to the instructions are not integers
4. If the number of arguments given to the instructions is higher or lower than expected

C. Karma & Personal Reflections

1. Karma Discussions
    a. Critter Comparison
    Besides the contest Critter, Skibur, we designed another critter named Gladiator, whose design is to try to stay in the corner or get together, turn rounds, and infect whatever Critter arrives.
    Advantage: As expected, in a battle where each side has more than 300 critters, Gladiator is likely to win if it can occupy a corner. We tested Gladiator with Skibur and Rover. Gladiator wins as soon as it can form a small group near any of the neighborhood on the map.
    Limitation: However, Gladiator is not favorable in a small amount. For example, in a 30 vs. 30 case, Gladiator is nearly impossible to win over either Skibur or Rover. This is largely because it is very hard to get together when most spaces are empty.

    For Skibur, however, the situation is opposite. It will win pretty fast if there’s less than 10x of the enemy on the field. However, when there is a significant number of enemies on the field, it is too hard to incorporate this condition into all the if statements we put into the .cri file, thus causing the critter to be vulnerable.
    b. Local behaviors vs. global behaviors
    For different critters, we have discovered different effects that local behaviors can cause to global behaviors.
        1) Skibur
        Based on the code design for Skibur, it tends to be mobilized so that it can be powerful. Whenever it finds the enemy, it will try to move away and through moving, infect more enemies. The picture below is from the battle with Rover, 100 Skibur vs. 300 Rover. We find that its local behavior of trying to move translates the whole picture to be dynamic.



2) Gladiator

Since Gladiator is more a static critter, in a sense that it tries to stay at a spot and infect others, it's local behavior of moving to a certain corner has caused some fascinating global behaviors.

In figure 1, there are a few Gladiators gather around near the top left corner and most enemy(Rover) is at the right side of the field. However, since Rover is moving around and trying to attack its enemy, the whole picture begins to be up left, and bottom right is staying still, while everything in between is more dynamic.

As the fest goes on, the situation translates into figure 2: as the Gladiators increases and Rover decreases, the right side of the picture becomes more dynamic. Gladiators' local behavior of infecting has caused mobility into the whole picture.

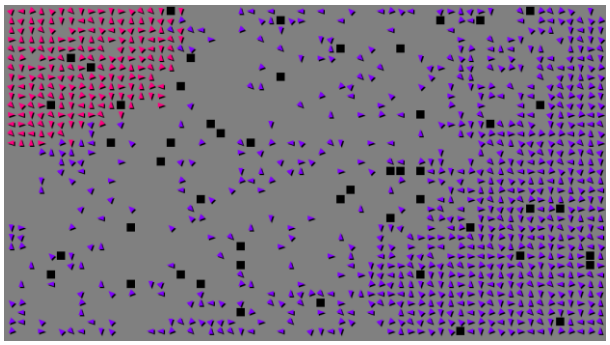


Figure 1

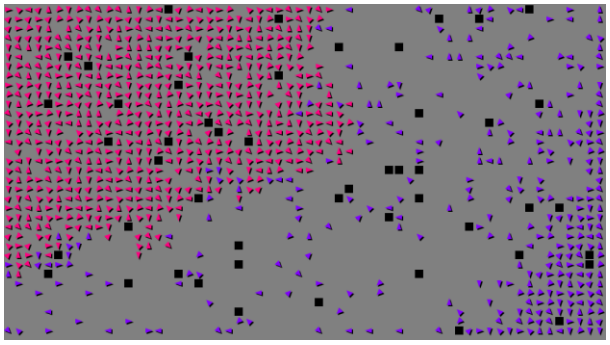


Figure 2

c. Critter compiler

2. What we learned from this project and future improvements

   For the one thing, this project greatly trained our logical thinking process. Unlike common programming language, the commands for critter are so primitive that we can't even use a loop structure, which is taken for granted by us. Instead, we frequently use *go n* statement to form a loop. This is particularly helpful in that we go through each step of the loop, which is an integral part of the thinking process.

   Besides, this is the first project that we work with someone else. This is a very refreshing experience for us. Throughout the two weeks, we have been getting more adaptive to each other and developed a better understanding. More details are under part 3 - pair programming experience.

   However, one thing that we definitely can improve is to start after devising a relatively practical code design. We started off by writing a lot of codes, which are hard to change after completion because it is hard to debug a large truck of codes. Next time, we will try to incorporate more ideas from XP, such as developing test cases as we write codes, finishing codes little by little. We will also try to start debugging earlier because we find some errors during debugging as well.

3. Pair programming experience

   A. Date and working time logs

Date	# of minutes	Items Accomplished
9/16/2017	Together: Zhaosong - 30 Atharva -30 Individual: N/A	Read through assignment descriptions together
9/18/2017	Together: Zhaosong - 25 Atharva - 30 Individual: N/A	Write *loadSpecies(String filename)* method and part of executeCritter
9/23/2017	Together: Zhaosong - 40 Atharva - 55 Individual: Zhaosong - 60 Atharva - 40	Think about the logic of critter commands on our own first, and then get together to write out command lines in .cri file
9/25/2017	Together:	Implement Critter and

	Zhaosong - 20 Atharva - 40 Individual: Zhaosong - 20 Atharva - 30	CritterSpecies classes; discussion for JUnit testing
9/27/2017	Together: Zhaosong - 60 Atharva - 70 Individual: N/A	JUnit Testing class write out; write parts of report together through Google Docs
9/28/2017	Together: Zhaosong - 180 Atharva - 180 Individual: Zhaosong - 120 Atharva - 60	Finishing the testing portion of the report, handling edge cases we had not considered before.

B. Experiences

Pair programming in project three is a new experience to both of us.

First, we find that it is easier for both of us to come up solutions or insights. For example, when designing the test harness, we have had a hard time understanding the relationship between Critter and CritterSpecies. Since in the original project these two object types are being taken care of by the simulator, we didn't develop too much understanding on the relationship between them. However, many thanks to pair programming, we were able to discuss our thoughts on it and finally agree on that Critter should inherit from CritterSpecies, which is reasonable because under a species can have different individuals.

Besides, pair programming gives each of us more incentives to work on the project. If working alone, we are likely to postpone the programming homework because of personal schedule. However, when working together, since we will periodically send messages to one another to ask if the one wants to work collaboratively. This simple text reminds us to worry about this project throughout the two weeks.

Regarding difficulty, we find transmitting one's thoughts to the other is slightly hard, especially when the idea becomes convoluted. For example, when working on the *executeCritter* method, one realizes that merely using an integer *i* to keep tracking line numbers is not satisfactory because once the method returns, the variable will be destroyed. However, it was somewhat hard to explain the whole idea for the other part to understand. To deal with this issue, we try to show by either writing something on paper or using a print command in Java to transmit ideas more clearly. Besides, as we work longer, we begin to understand each other much better!

Testing Methodology

The main idea behind the testing method was that it should check if the class 'Interpreter' functions correctly. That is, the loadSpecies method in the interpreter class should properly read the commands from the critter file, and the executeCritter method should iterate through the commands and produce the expected results. Since none of the methods from the interface were provided to us, the testing of this project turned out to be cumbersome and time-consuming. We created a new project Testproj specifically for testing. The project Testproject contains several classes and interfaces, explained in detail below.

Maintest: This class contains the main function, which constructs an interpreter object and calls the loadSpecies and executeSpecies methods.

Critter: Since the actual implementation of the methods in the Critter interface was not provided to us, we had to create a new class 'Critter' in the test project that contained all the methods that were initially present in the Critter interface. Some methods, such as get(), left(), right(), infect() and infect(int n) were never called since we slightly changed the code for the executeSpecies method (Explained in detail later on). Others, such as getNextCodeLine() at getCode() are necessary for the correct execution of the executeSpecies method.

CritterSpecies: This class contains functions for the name of the critter species, as well as the ArrayList of commands from the behavior file of that species.

CritterInterpreter: The CritterInterpreter interface in the test project was the same as the one used in the original project.

Interpreter: This is the Interpreter class, directly copied from our original project. However, this directly copied class is not testing-friendly. It contains certain methods such as hop, eat, ifempty, etc. which are not provided to us in the original project. Therefore, we had to implement all these methods separately in a Critter class.

It is important to note that this testing version of the *Interpreter* class is not the same as the assignment version of the *Interpreter* class. We had to make a few changes to the *executeSpecies* method in the test Interpreter to get the JUnit test working correctly. The testing version of *executeSpeicies* returns an integer instead of void. We had to make this change so that we could call the *executeSpecies* method in the JUnit test class and compare it with what we expected was the correct integer to be returned. For example, whenever the critter.hop() method was called, we replaced that code with return 0. So now, whenever 0 was returned, we immediately recognized that the hop method had been called.

Why is changing the code justified in this scenario?

The actual implementation of the Critter method is never provided to us while working on this project. We must assume, therefore, that these methods all work perfectly well, even for all the edge cases (For example, calling hop() on a critter directly facing a wall will do nothing to change the position of that critter.) The primary function of the Interpreter is just to execute the commands provided to us in the Critter interface. We only need to check if *all the expected Critter methods are called in the expected*

order, the expected number of times. Therefore, wherever a method from the Critter interface is called in the code, *we can simply replace that line with a block of code that lets us know that the method was called*.

Implementation

Starting with the Interpreter file for the original assignment, we made the following changes:

INITIAL CODE	REPLACED WITH
c.hop(); return;	return 0;
c.left(); return;	return 1;
c.right(); return;	return 2;
c.infect(); return;	return 3;
c.infect(int n); return;	return 4;
c.eat(); return;	return 5;
No turn-ending instructions in the program (edge case)	return -1

We only changed the code in places where a turn-ending instruction was called. The executeCritter method only returns when a turn-ending instruction is called. Using this property, we can automatically check whether the sequence of returned integers matches the expected sequence that we know to be correct. Note that the expected sequence of integers is calculated by us logically on paper, and not by using any code.

Testing with JUnit 4

We created a new JUnit class in the Testing project called *Crittertest2*, which had the method *test*, that did all of the testing. The class had an Array *exp* that stored all the expected return values, in the expected sequence (calculated by us manually). The test method then created an Interpreter and called the *executeCritter* method n times, where we decided *n* as a reasonable number for that particular test case. The *assertequals* method checked whether each of the integer values returned by *executeCritter*, in order, matched the ones we expected. If all the integer values matched, the test passed. Otherwise, it failed.

Testing ifally, ifempty, ifwall, ifenemy

For all test cases, we considered the following static situation as the critter's surroundings. The getCellContents method in the Critter class was implemented according to the following table:

Enemy	**Empty**	**Wall**
Ally	**Critter (Facing the empty cell)**	**Wall**
Enemy	**Ally**	**Wall**

So for example, ifemtpy 0 and ifally 180 are true every time, and ifwall 270 and ifenemy 0 will obviously be false (considering the arrangement above). This arrangement of the surroundings was designed so that each of the 'if' instructions could be tested separately, both in cases where the statement was true, and the statement was false.

Testing the Registers:

For testing the registers, we created a new *registers* array in *Crttertest2* that held 10 integers (The expected values of each of the 10 registers, calculated by us manually on paper.) We could then use the assertequals method on each of the integers in this array and each of the corresponding actual register values (provided to us in the Critter class of the test project) to check if they were equal. Using this technique, we were able to check the write, inc r, dec r, ifgt, iflt, ifeq, add and sub-methods.

Black Box Testing:

Before creating the JUnit test class, we tried out some random input codes and checked if the outputs matched by having the program print out the instructions executed and manually examining the output stream to check if it was as expected.

White box testing:

To check whether each separate instruction worked, we gave the program some simple input test cases that had individual instructions. The input was targeted to check the specified instructions. To save time, we often tested similar directions in a single test case - for example (ifempty, ifwall, ifally and ifenemy) and (inc, dec). The inputs we used were such that every block of code in the Interpreter was executed at least once during testing.

Testing Exceptions:

Using JUnit, we were able to verify that our program threw the expected exceptions- for example, IOException whenever an invalid behavior file was specified, and NumberFormatException, whenever the string provided to the gotostepn method, was not a valid integer.

Selected Test Cases

Provided below are a selected number of test cases. All the sets of instructions to be tested are covered at least once.

Critter Behavior Code	**Expected Array of returned Integers and Expected Array of Registers**	**Instructions Tested**	**Number of times executeCritter() was called**	**Test Passed/ Failed**
right left infect infect +1 eat ifwall 270 5 ifempty 45 3 ifempty 0 1	ch ={2,1,3,4,5,2,1,3,4,5} registers = {0,0,0,0,0,0,0,0,0,0}	Right, left, infect, infect (int n), eat, ifempty(true), ifempty(false), ifwall(false).	10	Passed
ifenemy 225 +2 go +3 hop go 1 left go 1	exp={0,0,0,0,0}	ifenemy(true), hop, go n	5	Passed
ifally 45 +2 go +3 hop go 1 left go 1	exp = {1,1,1,1,1}	ifally(false), left, go	5	Passed
inc r10 inc r1 dec r2 add r3 r2 sub r9 r10 hop go 1	exp = {0,0,0,0,0} registers = {5,-5,-15,0,0,0,0,0,-15,5}	Inc, dec, add, sub	5	Passed
write r5 3 write r3 4 ifeq r3 r5 +2 go +2 hop left go 1	exp = {1,1,1,1,1} registers = {0,0,4,0,3,0,0,0,0,0}	Write, ifeq. go	5	Passed

ifangle 45 90 +3 hop go 1 left go 1	exp = {1,1,1,1,1} registers = {0,0,0,0,0,0,0,0,0,0}	ifangle	5	Passed

Strengths and Weaknesses in our Testing Methodology

Strengths:

Testing the code increased our confidence in its correctness. Since we used JUnit, we were able to automate the testing process. Instead of having the program print diagnostic statements and examining those statements to make inferences about the correctness of our program, we were able to have the JUnit test method check for accuracy automatically. Having an automatic checking mechanism shortened the time required for testing individual cases considerably.

Weaknesses:

1. Since the functions in the Critter interface were not available, we had to create a new project TestProj, especially for testing. Having a whole new project just for testing was cumbersome.
2. To use JUnit properly, we had to change our code for the Interpreter, as explained above. Although the change was very slight and perfectly logical, one could argue that changing the code that you are testing, just for testing, is not ideal.