CS 314H Algorithms and Data Structures — Fall 2017	Programming Assignment #3
Critters	Due September 29, 2017

In this assignment you will work in pairs to simulate critters that behave according to some well-defined set of instructions. Your task consists of three components. First, you will write a simple interpreter that reads a set of instructions and performs the appropriate behavior for one critter. Second, you will create a test harness and explain how you used it to test your interpreter. Third, you will design and submit one critter to participate in the annual CritterFest competition. Each team needs to submit one critter by the due date, but you will have additional time to revise your critters before the actual competition (we will provide details about the contest later).

This assignment requires a bit more code than the previous ones, so we suggest that you develop and test your code in pieces to minimize debugging time. **We strongly encourage you to read the submission directions carefully before you start working.**

1 The Simulated World

The simulated world is a 2D grid of squares, where each square can hold one critter. Each critter has a graphical representation, has a heading, belongs to a particular species, and behaves according to a set of species-specific instructions. After some random initial placement of critters, each critter takes a turn at performing some actions, such as moving to another square or eating another critter.

1.1 Terminology:

A *heading* is the direction a critter is facing. Measured in degrees, the possible headings are 0, 45, 90, 135, 180, 225, 270, and 315.

A *bearing* is an angular difference between two headings, measured in degrees clockwise. The possible bearings are 0, 45, 90, 135, 180, 225, 270, and 315.

An *enemy* of a critter is any critter of a different species.

An *ally* of a critter is any critter of the same species.

A *behavior file* is a file that contains the specification of a critter species' behavior. Similarly, the *behavior code* is the set of instructions in a behavior file that controls the species' behavior.

A *register* is one of ten registers, named `r1` though `r10`, that can hold an integral value. **Note:** If you do not name your registers as specified, your program will be incorrect. In particular, the `r` is part of the register name.

1.2 Species Programming Language

The behavior of each species is defined by a simple program that consists of the following instructions.

`hop`	The critter moves forward if the square that it's facing is empty.
`left`	The critter turns left 45 degrees to face a new direction.
`right`	The critter turns right 45 degrees to face a new direction.
`infect` n	If the square immediately in front of this critter is occupied by an enemy, the enemy critter is infected and becomes a member of the infecting critter's species (converting an enemy to an ally). The infected critter retains the same position and heading, but behaves like the infecting species starting at the n^{th} instruction. The parameter *n* is an optional parameter; if missing, the infected species begins executing its new program at the first instruction.

`eat`	If the square immediately in front of this critter is occupied by an enemy, the enemy critter is lightly sauteed and treated as a comestible. After this turn the enemy critter no longer exists, and this critter gains a spurt of energy, taking two actions every step instead of 1 for a modest duration; the duration can be found as a constant in the Critter interface.
`go` n	This instruction jumps to another line in the behavior code. If n is a number preceded by a '+' or '-' character, then the jump is a relative jump, and execution should continue at the current instruction +/- *n*. If *n* is a number with no prefix, then the instruction is an absolute jump, and execution should continue at the n^{th} instruction in the critter's behavior code. If n is a register number, then the instruction is an absolute jump to the line number in the register. **All instructions that perform jumps may specify the target line number using any of these notations.**
`ifrandom` n	This instruction randomly jumps to the n^{th} instruction half of the time and is a no-op the rest of the time.
`ifempty` b n	The first parameter, b, is a bearing. If the adjacent square at the specified bearing is unoccupied, the critter will jump to the n^{th} instruction.
`ifally` b n	If the adjacent square at the specified bearing is occupied by a critter of the same species, the critter will jump to the n^{th} instruction; otherwise, continue execution with the next instruction.
`ifenemy` b n	If the adjacent square at the specified bearing is occupied by a critter of a different species, execution will jump to the n^{th} instruction; otherwise, continue execution with the next instruction.
`ifwall` b n	If the critter is adjacent to the border of the world at the specified bearing (that is, there are no more squares in that direction), the critter will jump to the n^{th} instruction; otherwise, continue execution with the next instruction.
`ifangle` $b1$ $b2$ n	If the off-angle (difference in heading) between the executing critter and the critter at bearing *b1* matches bearing *b2*, the critter will jump to the n^{th} instruction; otherwise, continue execution with the next instruction.
`write` r v	Write the integer *v* into register *r*.
`add` $r1$ $r2$	Add the value of register *r2* to that of *r1* and store the result in *r1*.
`sub` $r1$ $r2$	Subtract the value of register *r2* from that of *r1* and store the result in *r1*.
`inc` $r1$	Increment the value of register *r1*.
`dec` $r1$	Decrement the value of register *r1*.
`iflt` $r1$ $r2$ n	If the value of register *r1* is less than the value of *r2*, jump to the n^{th} instruction; otherwise continue execution with the next instruction.
`ifeq` $r1$ $r2$ n	If the value of register *r1* is equal to the value of *r2*, jump to the n^{th} instruction; otherwise continue execution with the next instruction.
`ifgt` $r1$ $r2$ n	If the value of register *r1* is greater than the value of *r2*, jump to the n^{th} instruction; otherwise continue execution with the next instruction.

A critter executes any number of `if`, `go` or state-manipulation instructions without relinquishing its turn. That is, the instructions listed on this page do not terminate a turn. A critter's turn ends when it executes any of the following instructions: `hop`, `left`, `right`, `infect`, or `eat`. On subsequent turns, the critter resumes execution from the point at which its previous turn ended. If for any reason a critter tries to resume execution at a line number outside the

bounds of its program, it simply does nothing for the remainder of its turns.

Each behavior file contains the species name on the first line, followed by a sequence of instructions, where each instruction resides on a separate line. The program ends with a blank line. Any text after the blank line is ignored and can be used to provide comments about the Critter's behavior. The behavior file for the Flytrap species is given below:

```
Flytrap
ifenemy 0 4
left
go 1
infect
go 1

The flytrap sits in one place and spins.
It infects anything which comes in front of it.
Flytraps do well when they clump.
```

In addition to the Flytrap, we will provide you with the following behavior files.

`Food`	This critter spins in a square but never infects anything. Its only purpose is to serve as food for other critters.
`Hop`	This critter just keeps hopping forward until it reaches a wall. This critter is not very interesting, but it's useful for debugging.
`Rover`	This critter walks in a straight line until it is blocked, infecting any enemies that it sees. When it can't move forward, it turns.

2 Your Assignment

To get started with the construction of your interpreter, retrieve the provided files from the webpage, which also contains documentation for the API that you will be using. In the `species` subdirectory of the distribution you will find several example creatures, and you should place any creatures you create in this subdirectory as well (the simulation system will automatically load all creatures in this subdirectory).

You should implement a class called `Interpreter` that implements the provided `CritterInterpreter` interface. Class `Interpreter` will perform two tasks; the `loadSpecies` method will read critter behavior files, and the `executeCritter` method will execute critter behavior code, where the legal commands for behavior code are specified in Section 1.2. While you must implement the `loadSpecies` and `executeCritter` methods, you may define additional helper methods if you wish.

2.1 The `loadSpecies` Method

Before the critters can start battling in their simulated world, the information for each species of critter must be read from a behavior file. The `loadSpecies` method, which is called several times from elsewhere in the simulation system, loads a behavior file. This method takes a String argument specifying a behavior file, and should read the specified file and construct a `CritterSpecies` object containing the read information for that critter species.

The two most important elements of information in a behavior file are the critter species name and the behavior code. The `CritterSpecies` class is designed to record both pieces of information. The species name is stored as a String and the behavior code can be stored in any List class (ArrayList, LinkedList, and Vector are available in the java.util package). Exactly how you store the behavior code inside the `List` is up to you, but be aware that you will be using this representation during execution of behavior code, so make your storage easy to work with. The `loadSpecies` method should return a newly constructed `CritterSpecies` object containing the above two species elements.

Once you implement the `loadSpecies` method correctly, you can run the simulator and can see the game world populated with critters! However, the critters won't move until you finish the next method...

2.2 The `executeCritter` Method

As the critter simulation is running, the simulation system needs to know what actions each critter wishs to make. Every turn, each critter must select an action to perform by having its behavior code executed, and the `executeCritter` method handles this execution. `ExecuteCritter` takes as a parameter a `Critter` object, indicating the critter whose behavior code should be executed. This method should retrieve the critter's behavior code, using the `getCode` method of the `Critter` class, execute the behavior code, and subsequently call one of the `Critter` class' action methods, `hop`, `left`, `right`, `eat`, or `infect`. The action methods do *not* actually move, turn, eat, or infect any critters, they simply record what a critter's next action will be (used by the simulation system later). Once one of the action methods has been executed, `executeCritter` should finish up and return. One of the action methods should be called before `executeCritter` returns, otherwise that critter will forfeit its action!

Many of the commands in the behavioral language test whether anything is near a critter, i.e. `ifally` or `ifenemy`. The `Critter` class contains two methods, `getCellContent` and `getOffAngle` that you can use to implement those instructions. See the online documentation for detailed descriptions of those methods.

Each critter has its own set of 10 integer registers it can use to store information. The `Critter` class methods `getReg` and `setReg` allow your interpreter to read and write the contents of those registers.

Finally, your interpreter must be able to record where it last stopped executing behavior code for each critter. The `Critter` class methods `getNextCodeLine` and `setNextCodeLine` allow you to record this information.

Once you finish implementing this method, fire up `CritterApplication` and get those critters battling!

2.3 Make a Critter

Now, this exercise wouldn't be any fun if you didn't make any critters of your own. Make an interesting critter, *i.e.*, one that is more complex than the provided critters. Include in your report an explanation of your critter's strategy and how it performs against other critters.

Name your critter `lastname.cri`, where `lastname` is your last name. You can turn in additional critters (in file names of your choosing), but this one is the one that we will consider "yours" for the contest.

2.4 About Interfaces

You will notice that we don't provide source code for the simulation program, only documentation for the interfaces. The advantage of abstraction and encapsulation is that your interpreter need not depend on the simulation engine or graphical interface that it is plugged into. *We will run your interpreter with a different engine than the one we provide you.* However, if you implement the interfaces correctly, it will not matter what engine your code will be run on.

If you use the command line, you can build your system by invoking the compiler as follows: `javac -cp Critter.jar assignment/Interpreter.java` (The `-cp Critter.jar` tells the compiler to use the classes in the JAR, adding it to the class path). Run your code by running the class CritterApplication using a class path that includes the JAR and your build directory. If you use an integrated development environment, you will need to figure out how to add a JAR to the class path (also called the build path). For example, in Eclipse, you should be able to copy the JAR into your project and then right click it and select "Build Path > Add to Build Path".

Hint: It will be easier to write the interpreter if you work incrementally, implementing and testing just a few behavior instructions at a time. Write a simple critter behavior file to test those instructions (you may want to remove the example critters from the 'creatures' subdirectory until you've implemented all the instructions they use). When you've got those instructions working, implement more.

2.5 Testing

You may be tempted to use the simulated world to test your interpreter, but this simulator is in fact terrible for testing purposes. (Can you explain why?) Instead, you should create a test harness that allows you to control all inputs and to

monitor all outputs: Here, you probably want to consider the critter's location and environment to be part of the input, as well as the critter's next command. You can then systematically explore the input space and check for anomalies.

3 Bonus Karma Activities

- Part of the fun of this game is to think about how a critter's local behavior translates into global behavior when there are many critters. Please share in your report any interesting insights you have about these behaviors.
- Designing a combat-effective critter requires much thought and planning. Design many critters and analyze their relative strengths and weaknesses.
- The critter programming language is very primitive and short on features (for example, there are no proper looping constructs or structured control flow). One way to address this shortcoming without breaking compatibility is to implement a *Critter compiler*. Design a nicer language for critter control and implement a compiler that translates your nicer critter programs into plain critter programs. Your write-up should include a description of your language extensions and important design decisions, along with usage instructions and a few examples.

4 CritterFest

Since we require everyone to make a critter, the obvious question is, Who has the fiercest critter? We will conduct a critter contest after the submission deadline. We will not provide specific details about the format of the contest, so your critter needs to be generally survivable under varying map configurations, population densities, and so on.

5 Submission Deadline

As usual, your submission is due by 5:00pm on the due date.

Pair Programmers: Read these directions carefully.

Each team of pair programmers should submit a *single* report with *both* team members names on it. You may use either name for your contest critter (unless your name is Zhou, in which case you should use your partner's last name), and the electronic submission may be done from either name, but make sure that the files and report credit both partners.

Reports should include a log of the amount of time spent driving and the amount of time spent working individually, *e.g.*, X drives 1 hour; Y drives 45 minutes; X works alone for 1 hour, etc. Of course, pairs will ideally do the vast majority of their work together, but it's more important to be accurate than to give the appearance of being an ideal pair. Pairs should ideally write the reports together and take turns driving while writing the report.

Include in your report a section discussing your experiences with pair programming. Was it effective in producing correct code? How does it differ from working alone? What difficulties or issues did you encounter?

Source code: We only care about the source code that you wrote. If you submit other files from the original distribution they will be disregarded.

Report: In your report, be sure to discuss any important limitations and design decisions in your interpreter, and please document your critter and your experience with critter strategy.

Your Critter: Each team is required to design a critter. Put this in `lastname.cri`, where `lastname` is one of the team member's last name. **If your last name is Zhou, then please use your partner's last name.**

Acknowledgments. This assignment is a variation of one given by Robert Plummer. We thank Matt Alden for developing an earlier version of this software, Mike Scott and Ehren Kret for providing Java code to help us simulate critters, and Walter Chang for his improvements to this assignment. Finally, we thank Tres Brenan for making it fashionable for critters to eat one another.