# CS 2112 Fall 2022
## Assignment 5
### Interpretation and Simulation

Due: Thursday, November 17, 11:59PM
Draft Design Overview due: Tuesday, November 8, 11:59PM

This assignment requires you to implement

- an **interpreter** for the critter language introduced in the last assignment,
- a **simulator** that maintains a state of the execution environment and emulates the execution of programs, and
- a **console interface** for controlling the simulation and querying the state of execution.

In addition to implementing new functionality, you are expected to make sure that the functionality implemented for Assignment 4 works correctly. This may require fixing bugs in your code. However, the majority of the grades in this assignment will be on the new functionality.

## 1 Changes

- No changes yet!

## 2 Instructions

### 2.1 Grading

Solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand, is modular, and takes advantage of inheritance to maximize code sharing. A correct program compiles without errors or warnings and behaves according the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variable names and proper indentation. Public methods should be accompanied by Javadoc-compliant specifications. Class invariants should be documented. Other comments should be included to explain nonobvious implementation details.

### 2.2 Final project

This assignment is the second installment of the final project for the course. Read the Project Specification to find out more about the final project and the language you will be working with in this assignment.

### 2.3 Partners

You will work in a group of two or three students for this assignment. This should be the same group as in the last assignment.

Remember that the course staff is happy to help with any problems you run into. Read all Ed posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

### 2.4 Restrictions

Use of any standard Java libraries from the Java SDK is permitted. However, the use of a parser generator (e.g., CUP) is prohibited.

The release code contains all the new classes you should add to your existing code. You should follow the instructions in their Javadoc, so that our testing software can test your code.

### 2.5 Release

The release files are available on CMS. You should download them and incorporate them into your `critterworld` project. The folders correspond to packages in your project.

## 3 Design overview document

We require that you submit an early draft of your design overview document in advance before the assignment due date. The Overview Document Specification outlines our expectations. Your design and testing strategy might not be complete at that point, but we would like to see your progress. Feedback on this draft will be given promptly after the overview is due.

## 4 Version control

As in the last assignment, you must submit file `log.txt` that lists your commit history from your group.

Additionally, you must submit a file `a5.diff` showing differences for changes you have made to files you submitted in Assignment 4. Version control systems already provide this functionality.

## 5 Interpretation

The core of this assignment is implementing an **interpreter** for critter programs. An interpreter is a program that emulates the execution of programs written in some programming language. For example, the Java run-time system includes a **bytecode interpreter** that executes "bytecode" from Java class files.

Your interpreter will work directly on the AST generated by the parser from Assignment 4. It will interpret the rules by recursively evaluating the AST nodes representing conditions and expressions in the context of the current state of the critter and the state of the world. The current state of the critter and the state of the world are known as the **execution environment**. The interpreter executes rules until an action is taken. It also updates the critter's memory as described by the rules applied.

### 5.1 Loading new critters

To add a new critter to the world, the critter's initial state and program are read from a **critter file**. The critter files may contain blank lines and lines beginning with `//`, indicating comments. These lines should be ignored. Lines may be terminated either with just a newline character (`'\n'`) or a Windows-style `"\r\n"` sequence, and trailing whitespace is allowed on any line. Otherwise, the format of a critter file is as follows:

```
species: <name>
memsize: <memory size>
defense: <defensive ability>
offense: <offensive ability>
size: <size>
energy: <energy>
posture: <posture>
<program>
```

The species name `<name>` is a string. It is recorded for identification purposes, but is not otherwise used for this assignment and has no effect on the critter simulation. The next six values specified in angle brackets are nonnegative integers. The first represents the number of memory locations of the critter and the rest

represent initial values for some of the memory locations. Following these values are the critter rules. These are given in the syntax described in the Project Specification. The critter rules should be parsed with your parser from Assignment 4. An example of a critter file is given in the `example` directory. A valid critter file must have these elements occurring in this order. Except for syntax errors generated while parsing the critter rules, any anomalies discovered when reading a critter file should result in a warning message to the user and a default value supplied if appropriate, but execution should proceed. In addition to specifying a critter file to load, the user should be able to specify the number of such critters to be added to the world. These critters are placed at randomly chosen legal positions in the world: that is, not on top of a rock, food, or another critter.

### 5.2 Interpreting critter rules

You will need to implement the recursive algorithm described in the Project Specification to decide which action to take using the evaluated AST. You will also need to use your AST mutation code from the last assignment to implement mating and budding.

## 6 Simulation

A **simulator** keeps track of the state of the world and all the critters and other artifacts in it. Your simulator will load the initial state of the world from a file.

### 6.1 Loading world definitions

The initial state of the world is given in a **world file**, which may contain blank lines and lines beginning with `//`, indicating comments. These lines should be ignored. The first two lines of the world file have the following format:

```
name <world name>
size <width> <height>
```

The `<world name>` parameter is a string specifying the name of the world, which should be printed out when the world is loaded. The `<width>` and `<height>` parameters specify the width and height of the world. Each subsequent line must have one of the following three forms, which specify where to place a rock, food or a critter:

- `rock <column> <row>`
- `food <column> <row> <amount>`
- `critter <critter file> <column> <row> <direction>`

You are not required to check for objects being placed on the same hex or on hexes outside of the world, although you are encouraged to do so. All critter files must be in the same directory as the world file. This means that if the world file is located at `/home/bob/world.txt`, and that file contains the line `critter alice.txt 0 0 0`, then the critter file is located at `/home/bob/alice.txt`. Two methods that may be useful for finding critter files are `java.io.File.getAbsoluteFile()` and `java.io.File.getParent()`. An example world file is given in `world.txt`. As with critter files, any anomalies discovered when reading a world file should result in a warning message to the user and a default value supplied if appropriate, but execution should proceed.

### 6.2 Simulating the world

You will need to implement a model that keeps track of the state of the world: its dimensions and contents, critters and their states, etc., as described in the Project Specification. The world will be able to advance time steps, update the state of the world, and allow each critter to execute its rule set in each time step.

```
#   F   -   -
  -   -   -
-   -   -   #
  -   -   -
-   1   -   -
  -   -   -
#   -   -   -
  -   -   -
-   5   -   -
```

**(a)** An ASCII-art map of the world

```
(0,8)   (2,8)   (4,8)   (6,8)
    (1,7)   (3,7)   (5,7)
(0,6)   (2,6)   (4,6)   (6,6)
    (1,5)   (3,5)   (5,5)
(0,4)   (2,4)   (4,4)   (6,4)
    (1,3)   (3,3)   (5,3)
(0,2)   (2,2)   (4,2)   (6,2)
    (1,1)   (3,1)   (5,1)
(0,0)   (2,0)   (4,0)   (6,0)
```

**(b)** Coordinates in an ASCII-art map

**Figure 1:** The structure of ASCII-art maps

## 7 User interface

The `console.Console` class is provided to you. If you implement the `Controller` right, the command line interface should just work. It should support the following commands:

- `new`
  Start a new simulation with a world populated by randomly placed rocks.
- `load <world file>`
  Start a new simulation with the world specified in file `<world file>`. Your world initializer should read critter files associated with any critters specified in `<world file>`.
- `critters <critter file> <n>`
  Read the critter file `<critter file>` and randomly place `n` such critters into the world.
- `step <n>`
  Advance the world for `n` time steps.
- `info`
  Print the number of time steps elapsed, the number of critters alive in the world, and an "ASCII art" map of the world. The hex contents displayed in the map should follow these notations:
  - `-` for an empty space
  - `#` for a rock
  - *d* for a critter facing in direction *d*
  - `F` for food.

  Figure 1(a) shows an example ASCII-art map for a world with 7 columns and height of 9. The columns of this map corresponds to the columns of the world, and adjacent columns are staggered by one line. Figure 1(b) shows the `(column,row)` coordinates corresponding to various positions on the example ASCII-art map.
- `hex <column> <row>`
  Print a description of the contents of the hex at coordinate `(column,row)`. If a critter is present, print the following as a description of the critter:
  - its species
  - the contents of at least its first seven memory locations
  - its rule set, using the pretty-printer from Assignment 4
  - the last rule executed

  If food is present, print the amount of food.

## 8 Written problems

**1.** A **bag** is a collection that allows duplicate elements. The code below partially implements a bag abstraction.

```
1  /** A Bag is an unordered collection of elements (of type T). Elements
2   *  are non-null and may be duplicates of other elements in the bag. */
3  public class Bag<T> implements Collection<T> {
4      private static class Node<T> {
5          T elem;
6          int count;
7          Node<T> next; // May be null.
8          // Invariant: count is positive and nodes starting from
9          // next form a null-terminated linked list.
10
11         public Node(T x) {
12             elem = x;
13             count = 1;
14             next = null;
15         }
16     }
17     // Class invariant: Every node in the list starting from 'head'
18     //  has a distinct n.elem.
19     // Representation: Each element n.elem is present in the bag the
20     //  number of times specified by n.count.
21     private Node<T> head = null; // null if bag is empty
22
23     /** Effect: Adds x to the bag. If x is already in the bag,
24      *  adds x again to the bag and returns true. Otherwise,
25      *  returns false. */
26     public boolean add(T x) {
27         Node<T> n = head;
28         if (n == null) { head = new Node<>(x); return false; }
29         while (!n.elem.equals(x) && n.next != null) {
30             n = n.next;
31         }
32         // At this point n != null, and if there is a node m in the list
33         // where m.elem equals x, then n==m.
34         // Otherwise, n is the last node in the list.
35         if (n.elem.equals(x)) {
36             n.count++;
37             return true;
38         } else {
39             n.next = new Node<>(x);
40             return false;
41         }
42     }
43 }
```

Read the code carefully and answer the following questions:

a) Consider a bag constructed by making $n$ calls to the method `add()`. What is the worst-case asymptotic time to construct the whole bag, as a function of $n$?

The code gives a postcondition for the while-loop. Let's construct the argument that the while-loop correctly achieves this postcondition.

b) Give a loop invariant for the while-loop that is strong enough to show its correctness.
c) Argue that the loop invariant is established at the beginning of the loop.
d) Argue that the loop invariant is preserved by each loop iteration.
e) Use the loop invariant to argue that the postcondition holds after the loop.

**2.** Write a critter program for a critter that walks in a growing hexagon spiral that, on an infinite world without any rocks, would eventually hit every hex. When it comes to food, it should eat the food. (**Hint:** the critter will need additional memory slots.)

3. Write a critter program for a critter that sits in one place until food appears within one hex of it. It then eats the food and moves to where the food was. While sitting, whenever it gets within 100 of its maximum energy, it tries to bud a child.

## 9 Overview of tasks

Determine with your partner how to break up the work involved in this assignment. Here is a list of the major tasks involved:

- Implement the interpreter for critter programs.
- Implement the state of the world and its critters.
- Implement the console interface and its communication with the world model.
- Develop a good test suite to ensure that the interpreter is implemented correctly.
- Solve the written problems.

## 10 Tips and tricks

**Modular design** Think carefully about how to divide this programming assignment up into modules that separate concerns effectively. For example, can you keep the interpreter code largely separate from the rules of the world simulation? Can you express the rules of the world simulation simply and in a localized way that makes it clear they are correct? In Assignment 6, the console interface will be replaced by a graphical user interface (GUI), which will display information similar to the current command-line interface. Consequently, if your world model is properly decoupled from the user interface, you should be able to substitute the GUI for the command-line interface without changing the world simulation. This is the essence of the Model-View-Controller design pattern.

**File paths** Note that all files (critter and world definitions) should be specified by relative file paths from the project root. Make sure to write your relative file paths in an OS-agnostic way; that is, you should not be hard-coding in any back or forward slashes.

**Testing** Your testing strategy will be important for this assignment, and we expect you to put time into planning how you will test and to document your testing plan in your design document.

Testing the world simulation is difficult without a graphical representation. The main focus of this assignment is therefore on correctly interpreting critter programs, rather than on perfecting the world simulation. Our grading scheme will reflect this priority. We recommend that you work with small worlds and focus on testing each language construct in isolation and on testing individual critter actions. Make sure your `info` command prints out accurate ASCII-art representations of the world so you (and we) can tell that your code is correct.

It may be difficult to debug your implementation using only the output of the program as defined in the specification. We recommend adding additional diagnostic functionality so that you can see, for example, why each rule is chosen or not chosen during the evaluation. We also recommend developing unit tests for each language construct. For example, you want to be sure that all the sensors produce the right values and all the actions do what they are supposed to. Testing correctness fully might be challenging to achieve by only running the simulation, so think about what other test harnesses would be helpful. Time spent making viewing and testing as easy as possible will be well worth it. If you put all your tests in `src/test/java`, Gradle will run them for you every time you run the Gradle `build` task and print a report, which you can find in the Gradle build folder. You should be sure to test:

- loading a full critter file
- generating a new world
- loading a full world file

- stepping a single critter
- stepping multiple critters
- printing the ASCII-art world
- that the Spiral Critter travels in a spiral path.

This list is by no means exhaustive, but rather offers a few key milestones. Your full suite of tests should be more thorough.

## 11 Submission

You should submit these items on CMS:

- `overview.txt/pdf`: Your final design overview document for the assignment. It should also include descriptions of any extensions you implemented.
- A `zip` file containing these items:
  - **Source code**: You should include all source code required to compile and run the project. Source code should reside in the `src/main/java` directory with an appropriate package structure.
  - **Tests**: You should include code for all your test cases. These should be in in `src/test/java`, separate from the rest of your source code. Subpackages are permitted.
  - **External libraries**: If you imported any external libraries via Gradle, include your `build.gradle` file.

  Do not include any `.class` files.
- `log.txt`: A dump of your commit log from your version control system.
- `a5.diff`: A text file showing diff of changes to files that were submitted in the last assignment, obtained from the version control system.
- `bag.txt`: This file should contain your solution to the bag written problem.
- `spiral.txt`: This file should be a plain text file containing your solution to written problem 8.1 (spiral critter program) and nothing else. It should be possible to load and parse the file with the `ParseAndMutateApp` program from A4. It does not need to include the rest of the metadata included in critter files - just the program.
- `eat-and-bud.txt`: This file should be a plain text file containing your solution to written problem 8.2 and nothing else. It should be possible to load and parse the file with the `ParseAndMutateApp` program from A4. It does not need to include the rest of the metadata included in critter files - just the program.