CS 2112 Fall 2026

Assignment 4

Interpretation and Simulation

Due: Thursday, November 5, 11:59PM
Midpoint checkpoint due: Monday, November 2, 11:59PM
Design Document due: Tuesday, October 27, 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 3 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.

Updates

- None yet; watch this space!

1 Instructions

1.1 Grading

Solutions will be graded on design, correctness, testing, and style. A good design makes the implementation easy to understand and maximizes code sharing. A correct program compiles without errors or warnings and behaves according to the requirements given here. A good test plan ensures good coverage of features and edge cases. 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.

Finally, like in the previous assignments, a small portion of your assignment grade will be based on a short, in-person **oral evaluation** with a course staff member after you submit your work. In this evaluation, we will ask you a few questions about your solution. Note that we will not be testing for correctness; rather, we will look to see whether you fully understand and can explain the work you submitted. As in Assignment 3, oral evaluations will be individual for this assignment. You are expected to understand and be able to explain your team's work as a whole well, and the parts you wrote yourself entirely.

1.2 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.

1.3 Teams

You will work in groups of two or three for this assignment. This should be the same group as in Assignment 3.

After each assignment, you will be asked to fill out a brief survey on CMSX providing peer evaluations for your teammates, and they will do the same for you. Peer evaluations will be reviewed by course staff

and become a component of your grade. More information about the peer evaluation procedure can be found on the course website.

Outside your group, remember that the course staff are happy to help with 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.

1.4 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 that you should add to your existing code. You should follow the instructions in their Javadoc, so that our testing software can test your code.

1.5 Release

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

You will also need to make the following changes to your project `build.gradle` file:

- Underneath the **application** block (the block with `mainClass` inside), add the following code to include console jar for A4:

```
// console jar for A4
jar {
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
    archiveBaseName.set('Console')
    manifest {
        attributes 'Main-Class': 'console.Console'
    }
    from(sourceSets.main.output) {
        include '**'
    }
}
```

- Inside the **application** block, change `mainClass` so that it reads

```
mainClass = 'console.Console'
```

1.6 Generative AI

Like in the previous assignments, you are allowed to use generative AI for this assignment, but **only** to help with writing test cases. Note that you **must** still write some test cases by hand, and that in our experience, test cases written by GenAI are not up to the standards expected by this course. Remember that course policy requires that you document all prompts given to GenAI in your final overview document. Furthermore, if you choose to use AI to help with testing, you must distinguish the tests generated by AI in your code, either by separating them into different files or by leaving a comment flag above each test (such as `// NOTE: AI-GENERATED`). If you choose not to use AI to help with testing, a sentence in your final overview document that states this decision will suffice. Basic code completion in IntelliJ is still permitted without attribution. Consult with the course staff if you have specific questions.

2 Design overview document

As with A3, 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.

These are key topics to cover in your design overview document:

- What are the key data structures you will use for this assignment to simulate the world? What are the core classes and abstractions you will implement?
- What are the key algorithms you will need? Which ones will be challenging to implement, and why?
- What will be your testing strategy for this assignment?
- What will be your implementation strategy, and how will you go about dividing responsibilities between the group members?
- What are some design patterns that would be useful for helping to separate concerns such as interpretation and simulation in your implementation?

We also highly recommend that you read section 11 regarding other tips and tricks for the assignment. We have a small portion dedicated towards good modular design which would be beneficial to read before starting the assignment. We recommend spending some time to answer the questions in that section in your design overview document.

3 Version control

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

Before you do anything else, we highly recommend you create a tag in your GitHub repo with the state of your final A3 submission, so that you can easily reference it in the future. From the right hand panel of your GitHub repo, choose the "Create a new release" link. Then, give the release a title, and click the dropdown above that says "Choose a tag". Give your tag a memorable name like a3 and choose "Publish release". This will mark this location in time on your repo with the tag name for easy reference.

You must submit a file named `a4.diff` showing differences for changes you have made to files you submitted in Assignment 3. To generate the diff file, you can then use `git diff a3 main > a4.diff` to generate a file named `a4.diff` with all the changes between your `a3` tag and the current state of your `main` branch.

4 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 3. 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.

For this assignment, do **not** implement the `smell` sensor. Instead, the expression `smell` should always evaluate to `0`. The full implementation of `smell` will come in a future assignment.

4.1 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, as described in the previous assignment. Lines may be terminated either with just a line feed 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 3. 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.

4.2 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 budding.

5 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.

As mentioned in section 4, the `smell` sensor should always evaluate to `0` for this assignment.

5.1 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.

5.2 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.

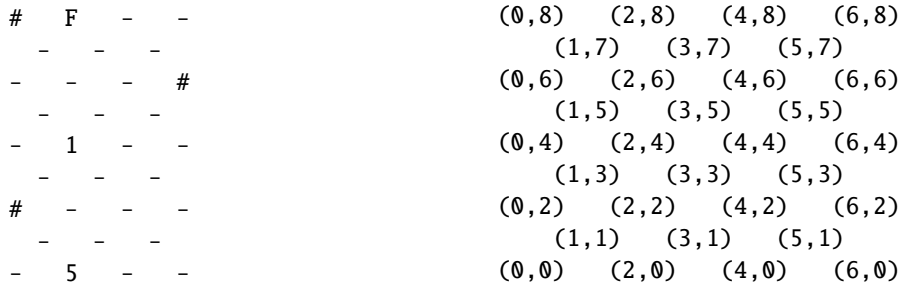
6 User interface

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 food is present, print the amount of food. 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 3
  - the last rule whose command was performed

The console obtains its controller from `ControllerFactory.getConsoleController()`. This method must always return a working controller; it is what the console and our tests use. The console also accepts the command-line option `--disable-mutation`, which turns off random mutation of critters produced by



(a) An ASCII-art map of the world

(b) Coordinates in an ASCII-art map

Figure 1: The structure of ASCII-art maps

budding. Supporting this option is optional, but we recommend it: a reproducible simulation is much easier to debug. See the comments in `ControllerFactory` for what is needed.

7 Adjustable Priority Queue

While the full implementation of `smell` is not a portion of this assignment, we ask you to implement a data structure that will be used as part of `smell` in a future assignment. The end goal will be to use an adjustable priority queue to run Dijkstra's single-source shortest path algorithm over the graph of possible critter moves. However, Java's builtin `PriorityQueue` does not support efficient updates to the priority of elements already inserted. As a result, you will instead make your own priority queue.

Complete the class `BinaryHeap` in `src/main/java/heap` that implements the `AdjustablePriorityQueue` interface. As a hint, you will need an additional data structure (in addition to a binary heap) to be able to look up the index of any item in expected $O(1)$ time. Don't forget to keep this second data structure updated every time you modify the heap; checking the invariant that connects the data structures may save you a lot of debugging. A second hint is to unit-test your implementation of this interface extensively. Trying to implement algorithms that rely on a buggy data structure is a recipe for disaster.

At the beginning of each method, you must provide a brief justification of the method's run-time complexity. A short comment will suffice. The methods in your class must adhere to the required time complexities provided in the interface.

8 Testing

A portion of your grade will be based on the quality and exhaustiveness of your test cases. Due to the open ended nature of this assignment as well as the complexity of the tasks, small mistakes can often mean that large amount of test can fail from our side. As a result, it is paramount to design an exhaustive testing strategy and unlike the previous assignment, we do not provide a sample test plan. You **must** design your own test plan for this assignment of similar size and scope.

As a starting point, we recommend that you work with small worlds and focus on testing each language construct in isolation and on testing individual critter actions. This also includes testing language constructs from both the interpreter side and the simulation side; whether our interpreter accurately output that critter will walk versus actually simulating that walk are two different problems but both require isolation of the language constructs. 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. Additionally, it may be difficult to debug your implementation using only the output of the program as defined in the specification. As a result, it may be useful to add debugging flags that print out why each rule is chosen or not chosen during the evaluation.

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
- interpreting critter programs with a single instruction
- simulating critter programs with a single instruction
- stepping a single critter with more complex rules
- stepping multiple critters in a large world
- 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.

9 Written problems

1. Give a loop invariant for the following piece of code, which finds both the minimum and maximum value of an even-length array a while using only $1\frac{1}{2}$ comparisons per element. Justify your answer by showing Establishment, Preservation, Postcondition, and Termination.

```
1  // Precondition: a.length is even
2  int max = Integer.MIN_VALUE,
3     min = Integer.MAX_VALUE,
4     i = 0;
5  while (i < a.length-1) {
6     int x,y;
7     if (a[i+1] > a[i]) {
8        x = a[i];
9        y = a[i+1];
10    } else {
11       x = a[i+1];
12       y = a[i];
13    }
14    max = Math.max(max, y);
15    min = Math.min(min, x);
16    i += 2;
17 }
```

2. 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.
f) Use the loop invariant to argue that the loop terminates.

3. 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. It should only eat the food directly in its path. (**Hint:** the critter will need additional memory slots.)

4. 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.

10 Overview of tasks

Determine with your partner(s) 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 controller 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.

11 Tips and tricks

Build a solid base We recommend that you first get the `new` and `info` commands working on an empty world, and then make sure that world files containing rocks, food, and critters are loaded with everything in the correct hex, before you implement the interpreter. All of our tests rely on the world being loaded and displayed correctly, and the midpoint checkpoint (§12.1) checks exactly this.

Modular design It is especially important to think about properly designing your classes and how thinking about how they interact with each other. Without proper implementation of the MVC design pattern, teams will likely have to refactor their code. In fact, it is important enough that you **must** mention it in your design document to get full credit. The purpose of this section is to provide some insightful questions that will force you to think about proper class design.

1. Can you keep the interpreter code largely separate from the rules of the world simulation?
2. Can you express the rules of the world simulation in code simply and with separation of concerns so that it is obvious they are implemented correctly?
3. What classes should modify the internal values of critters and worlds?
4. How can we enforce certain objects only being able to read values in from critters and worlds, but not being able to modify them?

Think carefully about the questions provided above and how to solve the problems present above effectively will make it easier to succeed in future assignments. For example, in A5, 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.

12 Submission

12.1 Midpoint checkpoint

For this assignment, we ask that you submit a snapshot of your code in advance of the due date. By Monday, November 2, submit a `zip` file containing your `src/main/java` directory to CMSX, with the same package structure as the final submission. This checkpoint will not factor into your final grade for the assignment, but consider it an opportunity for a free first round of grading — it is for your benefit. We will run our tests on your snapshot and return the results to you promptly, so that any problems with the basics are found before the deadline. The tests take no time steps; they check that your code can start a new world, load a world file with rocks, food, and critters on the correct hexes, print the ASCII-art map of Figure 1, add critters with the `critters` command, and handle a missing world or critter file without crashing. Your interpreter and simulator need not be working at that point.

12.2 Final submission

You should submit these items on CMSX:

- `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.
- `a4.diff`: A text file showing diff of changes to files that were submitted in the last assignment, obtained from the version control system.
- `written.txt/pdf`: This file should contain your solution to the written problems.
- `spiral.txt`: This file should be a plain text file containing your solution to written problem 3 (spiral critter program) and nothing else. It should be possible to load and parse the file with the `ParseAndMutateApp` program from A3. 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 4 and nothing else. It should be possible to load and parse the file with the `ParseAndMutateApp` program from A3. It does not need to include the rest of the metadata included in critter files, just the program.