CS2112—Spring 2014

Assignment 5

Interpretation and Simulation

Due: Thursday, April 10, 11:59PM
Overview draft due: Tuesday, March 25, 11:59PM

In this assignment you will implement the critter world simulation as described in the project specification. This involves writing an interpreter for the critter rule programming language and using this interpreter to determine critters' actions during the simulation. The program will need to keep track of the state of the simulated world and of all the critters in it. Your program will be able to load both the initial state of the world and the definitions of new creatures to be loaded from files. In this stage of the project, you will use a console interface to run the world simulation and print information about the state of the world. We will add a graphical user interface in the next assignment.

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, for this assignment, we will place significantly higher weight on the interpreter and the simulation than on the fault injection capability from Assignment 4.

0 Changes

- Added a pointer to an example critter file. (4/1)
- Should print at least the first eight memory locations, not the first nine. (4/2)

1 Requirements

1.1 World Simulation

You will implement a model which keeps track of the state of the critter world: locations and their contents, critters and their states, etc. as described in the project specification. The world will be able to advance time steps, allowing each critter in each time step to take the action the critter's rule set specifies.

1.2 Simulating critter rules

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. (As a Java program runs, frequently run code is converted on the fly to machine code that the processor understands directly.)

Your interpreter will work directly on the AST generated by the Assignment 4 parser. 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 world state. It will execute rules until the action to use is decided and return that command. It will also update the critter's memory as described by the rules applied.

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 call your AST mutation code for the implementation of mating and budding.

We are providing an example AST implementation with this assignment, including code for mutation. You are not required to use it, and if you do choose to use it, you are allowed to make any changes to it. Even if you choose not to use our AST implementation, it will probably be helpful to look at the stubs for `eval` to determine how to interpret your own AST.

1.3 Loading new critters

Your program should be able to load a rules file and add new critters to the world using those rules. This requires integration with your parser from Assignment 4. In addition to specifying a rules file to load, you should also allow the user to specify how many critters to create using those rules; that number of critters are then placed at randomly chosen legal positions in the world: that is, not on top of a rock or other critter.

The syntax of a critter file is as follows. It should begin with a specification of some of the first few memory locations in the following format:

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

Each of the values specified is an integer. You can assume the attributes appear in this order, but you may choose to be more flexible. You may choose to support additional attributes too.

Following this section of the input file, the critter rules should appear, in the syntax described in the project specification. An example of a legal critter file may be useful.

1.4 Loading world definitions

Your program should be able to load a world from a text file. Each line in the text file will have one of the following forms:

1. `rock <row> <column>`
2. `critter <critter_file> <row> <column> <direction>`

You are not required to check for objects being placed on the same hex or on hexes outside of the game world, although you are encouraged to do so.

See `world.txt` for an example world specification.

1.5 Console Interface

We provide a console interface for you to implement. You should be able to handle the following commands:

- `new`
  Start a new simulation with a world populated by randomly placed rocks. Your program should automatically read the file `constants.txt` to determine the world parameters.
- `load` ⟨*world_file*⟩
  Start a new simulation with the world specified in file ⟨*world_file*⟩. If there are any critters specified in `world_file`, your program will open and parse their associated rules files. Your program should automatically read the file `constants.txt` to determine the world parameters.
- `critters` ⟨*critter_file*⟩ ⟨*n*⟩
  Load the critter definition from the file named by ⟨*critter_file*⟩ and randomly place *n* critters with that definition into the world.
- `step` ⟨*n*⟩
  Advance the world *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 columns of this map should correspond to the columns of the world, and adjacent columns should be staggered by one line. The figure to the left shows which (column, row) coordinates correspond to which positions on the map, and the figure to the right shows an example map of the same size. Use the following codes for hex contents: - = empty space, # = rock, Cn = critter facing in direction n, F = food, and Gn = critter facing in direction n + food. For example:

```
        (1,5)       (3,6)       (5,7)              -   -   -
(0,4)        (2,5)       (4,6)        (6,7)    #   F   -   -
        (1,4)       (3,5)       (5,6)              -   -   -
(0,3)        (2,4)       (4,5)        (6,6)    -   -   -   #
        (1,3)       (3,4)       (5,5)              -   -   -
(0,2)        (2,3)       (4,4)        (6,5)    -   C1  -   -
        (1,2)       (3,3)       (5,4)              -   -   -
(0,1)        (2,2)       (4,3)        (6,4)    #   -   -   -
        (1,1)       (3,2)       (5,3)              -   -   -
(0,0)        (2,1)       (4,2)        (6,3)    -   G5  -   -
```

- `hex <`*c*`> <`*r*`>`
  Print a description of the contents of the hex at column *c*, row *r*. If a critter is present, print the contents of that critter's first eight memory locations (at least), the critter's rule set (using pretty-printing from Assignment 4), and the last rule executed by that critter. If food is present, print the amount of food.

2 Advice

Keep in mind that in the next assignment, the console interface will be replaced by a graphical user interface. Fortunately, the graphical user interface will display much of the same information as the current interface, so if you design your world model to be properly decoupled from the user interface, you will be able to build your new interface without changing the world simulation. This is the beauty of the model-view-controller design pattern.

Also, we realize that testing the world simulation will be difficult without a graphical representation. The main focus of this assignment is the critter program interpretation rather than fixing every last bug in the world simulation, and we will grade accordingly. We recommend that you work with small worlds and focus on testing individual critters' actions.

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 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. This may be challenging to ensure by just running the world.

3 Programming tasks

You will want to figure out with your partner how to break up the work involved in this assignment. To get you started thinking about this, here are some of the major tasks involved:

- Implementing the recursive interpretation of the critter's program AST
- Implementing the state of the critter world and its critters.
- Implementing the console interface and its communication with your world model.
- Developing a good suite of test cases to ensure that the whole critter language is correctly implemented.

Please get started early and plan with your partner how you're going to do this assignment. We have set a relatively early due date for the overview draft document to inspire you to do this.

4 Restrictions

You may use any standard Java libraries from the Java SDK. However, you may not use a parser generator.

5 Overview Draft

We are requiring you to submit an early draft of your design overview document by March 25. As usual, you may not be able to predict what your design and testing strategy will look like in full at that point, but we want to see how far you have gotten. We will aim to get you quick feedback on this draft.

6 Submission

You should compress exactly these files into a zip file that you will then submit on CMS:

- *Source code*: You should include all source code required to compile and run the project.
- *Tests*: You should include code for all your test cases, along with the world and critter files you used for testing.
- `overview.txt/html/pdf`: This file should contain your overview document.

Do not include any files ending in `.class`.

7 Written problem

Consider the following simple code for sorting an array, where `swap(a,i,j)` swaps array elements in the obvious way:

```
/** Effects: Sort the elements of a into ascending order. */
void sort(int[] a) {
    int n = a.length;
    for (int i = 0; i < n-1; i++)
        for (int j = i+1; j < n; j++)
            if (a[i] > a[j]) swap(a, i, j);
}
```

1. How does the best- and worst-case asymptotic performance of this code compare to that of insertion sort? Why would you expect it to be slower in practice? Explain briefly.
2. How does the best- and worst-case asymptotic performance of this code compare to that of selection sort? Why would you expect it to be slower in practice? Explain briefly.
3. Give a loop invariant for the outer loop that is strong enough to show that the sorting algorithm works correctly. Give a clear argument for each of the three aspects of partial correctness: establishment, preservation, and the postcondition. Remember that the preservation argument can only rely on things being true if they are part of the loop invariant itself.