CS2112—Fall 2015
Assignment 5
Interpretation and Simulation

Due: Thursday, October 29, 11:59PM
Design Overview due: Wednesday, October 21, 11:59PM

This assignment requires you to implement an *interpreter* for the simple 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 the execution environment.

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. The majority of the grades in this assignment will be on the new functionality, however.

0 Changes

- 10/27: Added another karma problem.
- 10/18: Food syntax in `world.txt` clarified.

1 Instructions

1.1 Grading

Solutions will be graded on design, correctness, 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 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. Keep your code within an 80-character width. Methods should be accompanied by Javadoc-compliant specifications, and class invariants should be documented. Other comments may be included to explain nonobvious implementation details.

1.2 Final project

This assignment is the second part 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 Partners

You will work in a group of two 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 problems you run into. Read all Piazza 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. Use of a parser generator (e.g., CUP) is prohibited, however.

2 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 within 72 hours after the overview is due.

3 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.

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

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 execution environment. The interpreter executes rules until an action is taken. It also updates the critter's memory as described by the rules applied.

4.1 Loading new critters

To add a new critter to the world, the critter's initial state and rule set is read from a critter file. Each critter file begins with a line specifying the name of the critter's species, followed by a specification of some of the first few memory locations:

```
species: ⟨species name⟩
memsize: ⟨memory size⟩
defense: ⟨defensive ability⟩
offense: ⟨offensive ability⟩
size: ⟨size⟩
energy: ⟨energy⟩
posture: ⟨posture⟩
```

The species name is a string. It is recorded as an attribute of critters but is not otherwise used for this assignment; it has no effect on the critter simulation. Each of the other values specified in angle brackets is a nonnegative integer. You can assume the attributes appear in the specified order, but you may choose to be more flexible.

Following the initial state in the critter file are the critter rules, in the syntax described in the Project Specification. An example of a critter file is given in the `example` directory.

Reading critter files requires integration with your parser from Assignment 4. 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 other 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 mating and budding.

An implementation of the AST and the parser is provided with this assignment. You are not required to use it, but if you do, you are allowed to make changes, and you will have to provide your own mutation code to avoid a small penalty. Therefore, we recommend that you use our implementation as a guide to help fix your own code rather than as a replacement.

5 Simulation

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

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 world file description below assumes no existence of such lines.

The first two lines of the world file has the following format:

```
name ⟨world name⟩
size ⟨columns⟩ ⟨rows⟩
```

The `world name` parameter specifies the name of the world, which should be printed out when the world is loaded. The `columns` and `rows` parameters specify the number of columns and rows in the world. Every subsequent line must have one of the following three forms, which specify where to place a rock, food or a critter:

1. `rock` ⟨*column*⟩ ⟨*row*⟩
2. `food` ⟨*column*⟩ ⟨*row*⟩ ⟨*amount*⟩
3. `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.

An example world file is given in `world.txt`.

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, updating the state of the world and allowing each critter to execute the rule set in each time step.

6 User interface

A skeleton of the console interface is provided. You should implement the interface to support the following commands:

- `new`
  Start a new simulation with a world populated by randomly placed rocks. Your world initializer 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*⟩. Your world initializer should read critter files associated with any critters specified in ⟨*world_file*⟩. Finally, your world initializer should automatically read the file `constants.txt` to determine the world parameters.
- `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.

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

(a) An ASCII-art map of the world

```
(0,4)       (2,5)       (4,6)       (6,7)
      (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)
      (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)
```

(b) Coordinates in an ASCII-art map

Figure 1: The structure of ASCII-art maps

Figure 1(a) shows an example ASCII-art map. 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 eight 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.

7 Written problems

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 `sort` compare to that of insertion sort? Why would you expect `sort` to be slower in practice? Explain briefly.
2. How does the best- and worst-case asymptotic performance of `sort` compare to that of selection sort? Why would you expect `sort` to be slower in practice? Explain briefly.

3. Give a loop invariant for the outer loop of `sort` 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 assumptions that are part of the loop invariant itself.

4. Write a critter program that walks in a growing spiral that, on an infinite world without any rocks, would eventually hit every hex. When it comes to food, it should eat the food.

8 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:

- Implementing the recursive interpretation of critter programs.
- Implementing the state of the world and its critters.
- Implementing the console interface and its communication with your world model.
- Developing a good test suite to ensure that the critter interpreter is implemented correctly.
- Solving the written problems.

9 Tips

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 the next assignment, the console interface will be replaced by a graphical user interface, which will display similar information as the current interface. Consequently, if your world model is properly decoupled from the user interface, your new interface can be built without changing the world simulation. This is the beauty of the Model–View–Controller design pattern.

Testing. 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 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 works.

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.

You will likely need to design dozens of small tests, so running the tests will not be convenient. Time spent making testing as easy as possible will be well worth it. We recommend that you write a test harness that can run multiple tests, or all your tests, at once, and tells you if any tests failed. Your test harness could be based on JUnit, or could be a Java program, a bash or Python script, or an `expect` script.

Good things to remember to test include:

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

10 Karma

Karma questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

10.1 Extensions to the final project

Possible extensions include but are not limited to the following:

- Add support for selecting and directly controlling a critter.
  Karma: ●●●○○
- How fast can you make the critter interpreter run? Think about ways to avoid unnecessary computation.
  Karma: ●●●○○
- Write a critter program that makes the critter walk backward, `serve`ing food onto successive hexes it leaves behind, in amounts that form the sequence of prime numbers: 2, 3, 5, 7, 11, ...
  Karma: ●●●○○

10.2 Postfix calculator

Tired of parsing parentheses and throwing them away? Can we avoid parentheses? Without them and without operator precedences, the usual way of writing down arithmetic expressions becomes ambiguous. This usual way is known as the *infix* notation, where each binary operator is between its two operands.

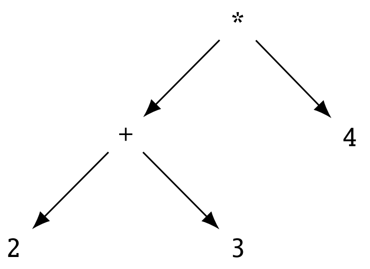

Figure 2: The AST for `((2+3)*4)` in infix notation and for `2 3 + 4 *` in postfix notation

As we learned from Assignment 4, abstract syntax trees eliminate the need for parentheses. For example, Figure 2 shows the abstract syntax tree for `(2+3)*4`. Still, pretty-printing this AST using inorder traversal requires reinserting parentheses at appropriate places. If postorder traversal is used instead, we obtain the following output: `2 3 + 4 *`. Pretty-printing using postorder traversal gives us the *postfix* notation, as each operator appears after all its operands have appeared. Postfix expressions need no parentheses, and evaluating them is simple: whenever a binary operator is encountered, we must have evaluated its two operands, which are the results of the two most recent evaluations that have not been used in some other evaluation. In our example, when + is encountered, we have seen 2 and 3 most recently, so the result of addition is 5. When * is encountered, we have seen 5 and 4 most recently, so the result of multiplication is 20.

HP calculators are famous for using postfix; some widely used programming languages work this way too, of which the best-known are FORTH and PostScript.

Implement the following method that evaluates a postfix expression:

```
/**
 * Evaluate the postfix expression given in a stream.
 * @param is An InputStream containing an expression
 * @return The result of evaluating the postfix expression
 * @throws SyntaxError if the expression is malformed
 */
public static int postfixInterpret(InputStream is) throws SyntaxError;
```

You should augment the above documentation to specify the behavior of the method in abnormal conditions. You may assume that each number in a valid postfix expression is nonnegative, and valid operators are *addop* and *mulop* in Section 5 of the Project Specification.

You may use the tokenizer, the parser, and the AST in your final project implementation to help solve this problem.

Examples:

- `2 + 3 + 4` throws a `SyntaxError`.
- `2 3 + 4 +` returns 9.
- `2 3 4 + +` returns 9.
- `2 3 4 * +` returns 14.
- `2 3 + 4 *` returns 20.
- `2 3 + 4 5 + *` returns 45.

- `2 2 2 2 2 2 * * * * *` returns 64.
- `1 2 * 3 * 4 * 5 * 6 * 7 *` returns 5040.

Karma: ● ● ● ○ ○

10.3 Josephus problem

The *Josephus problem* arose in the 1st century A.D.[1]. In July 67, Josephus, a Jewish historian, was trapped with forty of his companions in a cave after the Romans invaded a Jewish garrison. Declined to surrender to the Romans, the group committed a collective suicide using a method suggested by Josephus: the group stood in a circle, and every third person was killed. Josephus was the sole survivor of this process. He surrendered to the Roman forces and was later released.

It turns out that the original method was simple enough that Josephus was rumored to rig the "game." Henceforth, we will consider an extended version of the problem, where "every i^{th} person" varies in each round of killing. For example, suppose there are four people in the circle. In the first round of killing, the 5th person is killed. In the second round of killing, the 6th person is killed. In the third and final round of killing, the 7th person is killed. Using these rules, the game proceeds as follows:

1. The first person starts counting. The first person is also the 5th person to count, so he is killed.
2. In the second round, the second person, who was next to the first person, continues counting. He is also the 4th person to count. Now, the fourth person is the 6th person to count, so he is killed.
3. In the third round, the second person, who was next to the fourth person, continues counting. He is also the 3rd, 5th, and 7th person to count and hence killed. The third person is left standing and is the winner of the game.

Implement the following method to determine the winner of the extended Josephus problem:

```java
/**
 * Determine the winner of the extended Josephus problem.
 * @param n The number of people at the beginning of the game
 * @param counts An array of length {@code n}-1, where {@code counts[i]} is a
 *              positive integer indicating the number of people to count
 *              in round {@code i}+1 before a person is killed
 * @return An integer between 1 and {@code n} indicating the last person
 *         standing
 */
public static int extendedJosephusSurvivor(int n, int[] counts);
```

What is the asymptotic running time of your algorithm? Include this in your method documentation.

Examples: In the following, the first number indicates `n`, and the numbers in `[]` indicate the content of `counts`.

- `6, [1, 1, 1, 1, 1]` returns 6.
- `5, [2, 2, 2, 2]` returns 3.

[1] History taken from Wikipedia.

- `4, [5, 6, 7]` returns 3.
- `3, [333, 111]` returns 2.
- `2, [2014]` returns 1.

Note: This problem can be simulated to determine the survivor in $O(n + C)$, where C is the sum of all the entries in `counts`. There is a faster way to solve this problem, and we are looking for such a solution. Please do not submit the naïve solution using simulation.

Karma:

11 Submission

You should submit these items on CMS:

- `overview.txt/pdf/html`: Your final design overview document for the assignment. It should also include descriptions of any extensions you implemented and of any **Karma** problems you attempted.
- A `zip` file containing these items:
  - *Source code*: You should include all source code required to compile and run the project. All source code should reside in the `src` directory with an appropriate package structure.
  - *Tests*: You should include code for all your test cases, in a package named `tests`, separate from the rest of your source code. Subpackages are permitted.

  Do not include any files ending in `.class`.
- `log.txt`: A dump of your commit log from the version control system of your choice.
- `a5.diff`: A text file showing diff of changes to files that were submitted in the last assignment, obtained from the version control system.
- `written_problems.txt/pdf`: This file should include your response to the written problems.