PEST: Polymorphic Ecosystem Simulation Testbed

For this project you will implement a simulation of a very simple ecosystem. Creatures of several types, specified below, will inhabit the system. The number of creatures in the system will vary. Most changes within the simulation are driven by a simulated clock; this simulation clock has no connection to the system clock. When the simulation clock "ticks" each creature currently in the ecosystem is notified and given the chance to update its state, according to the behavioral rules given below. The order in which the creatures update their state will determine how some subtle interactions play out. For consistency, we adopt the following rules. All plants are notified that a tick has occurred before any insects are notified. Plants are notified in the order they were created, and so are insects.

The ecosystem is envisioned as an infinite grid of spaces, rather like a chessboard with an x-y coordinate system imposed on the squares. The program will read a script file and carry out the commands it contains, writing all logged output to standard out. The name of the script file will be specified as a command-line argument to the program.

Creatures:

The ecosystem may contain creatures of several types, distinguished by behavior. For convenience, each creature will have a unique identifier, a name if you will. Each creature will have an energy level, represented by a nonnegative integer, and a location, represented by a pair of coordinates. Some types of creatures can move, and moving costs energy. Simply standing still also costs energy for some creatures, whereas others actually gain energy simply by existing. Some types of creatures eat other creatures under certain circumstances, acquiring some or all of the energy of the eaten creature. An eaten creature loses energy, sometimes some, sometimes all of its energy. If any creature's energy level drops to zero, it dies.

There are four species of creatures in the ecosystem. The first two are vaguely insect-like:

sipper A mildly hungry insect. If it is not feeding, a sipper moves one space on each tick of the simulation clock; a sipper follows the movement pattern NE-SE-S-W-W, repeating the pattern indefinitely. Moving from one square to an adjacent square costs a sipper 1 unit of energy.

If a sipper enters a square containing an edible plant, it will interact with the plant. The sipper will (on the next tick) begin consuming energy from the fruit. A feeding sipper will consume 5 units of energy per tick, or all the fruit's energy if it has less than 5 units, until the sipper has raised its energy level to 25 or the fruit's energy level reaches zero. At that point, the sipper stops feeding and will resume movement on the next tick, continuing its pattern from the point it stopped.

muncher A marauder. If it is not feeding, on each tick a muncher moves two spaces forward, then one space to its left, and then turns 90^{o} counter-clockwise; munchers are always created facing east. Moving from one square to an adjacent square costs a muncher 2 units of energy; turning costs no energy.

If a muncher enters a square containing an edible plant, it will interact with that plant, interrupting its current move sequence if necessary. The muncher will consume 12 units of energy, or the fruit's total energy, whichever is less. The muncher will then cease feeding, and resume moving on the next tick. The muncher continues its movement pattern from the point it was interrupted.

A muncher does not limit the total amount of energy it may consume, but if its energy total reaches 50 it will divide into two new munchers, one with 30 units of energy and one with the remainder, which will be at least 20. By convention these will be given IDs obtained by concatenating B and S, respectively, to the ID of their "parent". If the fruit still has energy, the offspring will attack it on the next tick.

The other two species are essentially plantlike:

fruit An edible plant. However, unlike all other types of creature, these gain one unit of energy per tick simply by existing. There is no upper limit on the amount of energy a fruit may store.

flytrap An inedible, carnivorous plant. Existing costs a flytrap one unit of energy per tick. If a sipper or muncher is unlucky enough to interact with a flytrap, the flytrap will eat that creature, consuming all its energy. This happens immediately, as soon as the creature enters the flytrap's square. There is no upper limit on the amount of energy a flytrap may store.

Plants never move from the square in which they were created. A square will never contain more than a single plant, mainly to simplify the simulation rules. However, a square may contain an arbitrary number of insectoids.

As a general rule, an insectiod will determine if there is an available plant when the insectoid first enters the square. So, if an insectoid wanders into a square holding a flytrap it will be eaten immediately. We will avoid the scenario in which a new creature is created in a square already containing another creature.

Mortality: creatures of all types can die. For some types, there are multiple ways to achieve this. For others, there is only one. Some creatures may run out of energy (or suffer other fates), and die, during their own update operation. Some creatures may be killed by another creature during that creature's update operation. In all cases, the simulation should result in the dead creature being removed as soon after its death as possible. Slight penalties may be assessed for delayed removal; larger ones will be assessed if a dead creature is not removed at all.

The system must keep track of all creatures that currently exist, which implies some sort of data structure. Moreover, in order to ensure that you achieve a properly polymorphic system, you are required to use a single data structure to keep track of all creatures (not separate structures for insectoids and plants).

Script File Description:

For a change, lines beginning with the character (`'#'`) are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the script file.

Each non-comment line of the script file will specify one of the commands described below. Each line consists of a sequence of "tokens" which will be separated by single tab characters. **Bold** text indicates commands or keywords that will be used verbatim. Tokens will never contain a tab character. Command words will always be followed by a single tab character. A newline character will immediately follow the final "token" on each line.

`create` [**`sipper`** | **`muncher`** | **`fruit`** | **`flytrap`**] `<name>` `<energy>` `<x>` `<y>`

Create a creature of the specified type, with the given name and initial energy, at the given coordinates, and add it to the (appropriate) list.

`tick` `<nticks>`

This causes the simulation clock to tick the specified number of times. On each tick of the simulation clock, each creature in the ecosystem is notified of a tick. Plants should be updated first, then other creatures. Within each of those categories (plant and other), the notifications should be done in the order the creatures were created.

`status` `<name>`

Logs the current status of the named creature (if it exists). The status report should include the creature's type, its name, its energy level, and its current location.

`exit`

This causes your program to log a status display for every existing creature, and then deallocate all dynamic memory and terminate. The script file is guaranteed to contain an `exit` command.

Legend: in the commands above:

`<name>`	a string which is a unique identifier for a creature
`<energy>`	a positive integer representing the initial energy of a creature
`<x>`	an integer representing the initial x-coordinate of a creature
`<y>`	an integer representing the initial y-coordinate of a creature
`<nticks>`	a positive integer representing the number of ticks to be simulated

Each command must be logged to standard out, along with an informative message indicating the results when the command was processed. Note that every command should produce some informative output. The output from each command should be delimited in some manner, similar to the parsing homework assignment.