CS2112—Fall 2014
Final Project
SEAL: Simulating Evolving Artificial Life
Version of October 26, 2014

The final four programming assignments for this course make up a single project that you will work on as part of a two-person team. In this project, you will simulate a simple world of animals ("critters") that wander around, eat food, reproduce, and evolve. This world will include a graphical visualization that enables a user to take control of individual animals. Different species can also interact in this simulated world. Because of evolution, there will be multiple species eventually even if initially there was only one.

Critters move around on a regular, hexagonal grid. Critters need energy to survive, because everything they do requires energy. They gain a little energy from the sun each time step as long as they are not moving. Critters may also attack each other; when critters run out of energy, they die and become food for other critters. If critters accumulate enough energy, they can reproduce.

The genome of a critter includes a *program* that determines what the critter does on each time step. The program consists of one or more rules defining what critters should do under certain conditions. When critters reproduce, the genome is copied from the parent or parents to the new critter, possibly with some mutations applied to some rule(s). This means that critter programs may change over time and perhaps evolve to make them more effective.

The critter simulation will be implemented as a networked Java service with a graphical front end. This design permits multiple users to view and interact with the same virtual world.

In summary, this project involves developing the following components:

- a simple parser and interpreter for the critter language
- a graphical user interface
- a distributed implementation of the system

0 Changes to the spec

The following changes have been made since the initial release.

- A program must have at least one rule. (10/26)
- Corrected appearance example, and added rotation matrices for the coordinate system. (10/25)
- Changed production for unary negation. (10/8)
- Changed how world size is specified to `COLUMNS` and `ROWS`. (10/8)
- Specified how operators associate. (10/8)

1 The world

The world that critters live in is a large array of hexagonal tiles called *hexes*. A given critter is at any moment located on one of these hexes and facing in one of the six possible directions, as shown in Figure 1. The world advances in discrete time steps. On each time step, a critter may

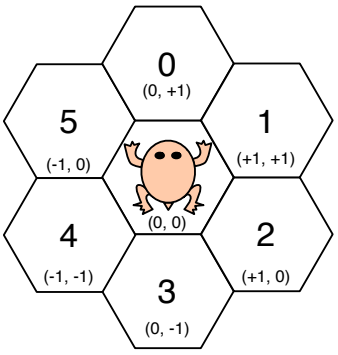

Figure 1: Hexes and directions

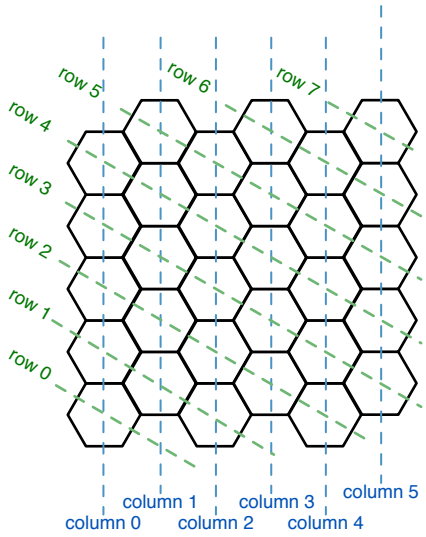

Figure 2: Rows and columns of hexes

perform one of several possible actions, or simply update its internal state and wait for the next time step while absorbing solar energy.

Hexes are either rock hexes or hexes that can contain something else. Rocks are inert obstacles that critters cannot move onto or over. A non-rock hex may be empty or may contain a single critter or some food, but not both. A hex cannot contain more than one critter.

Food is created on a hex when a critter dies there. Other critters may then eat the food to gain energy.

Hex coordinates Each hex is identified by a coordinate (c, r) where c is the column and r is the row on which the hex is located. Both c and r are nonnegative integers. Figure 2 shows the column and row coordinates of various hexes. The world has a fixed, roughly rectangular shape that is symmetric with respect to a 180-degree rotation. The lower-left (southwest corner) hex is at coordinate (0,0). Moving in one of the six possible directions changes the row and column coordinate of the critter. The corresponding adjustments to row and column coordinates are shown in Figure 1.

Some coordinates lie outside the world. A coordinate that lies outside the world acts in all ways as though it is a rock hex. Critters cannot fall off the edge of the world, and they see rock when they look off the edge. Figure 2 shows a very small world with `COLUMNS` = 6 and `ROWS` = 8. A coordinate (r, c) where $c \geq$ `COLUMNS` lies off the east edge of the world. A coordinate such as (1,0), where $2 \cdot r - c < 0$, lies off the south edge of the world. And a coordinate where $2 \cdot r - c \geq 2 \cdot$`ROWS`$-$`COLUMNS` lies off the north edge of the world. The world will be roughly square if `ROWS` is about 1.37 times as large as `COLUMNS`.

You might notice that there is a third potential "coordinate" we could specify, corresponding to lines of hexes slanting upward toward the right. We can define the "slice" s of a given hex as $(r - c)$, increasing as we move upward along a column or "northwest" along a row. Hexes in the lower right corner of the map have a negative slice. Using slices, we can compactly represent the

length of the shortest path between two hexes:

$$\max\{|\Delta c|, |\Delta r|, |\Delta s|\} = \max\{|\Delta c|, |\Delta r|, |\Delta r - \Delta c|\}$$

For those familiar with linear algebra, it is easy in this coordinate system to represent rotations around the origin, using 2×2 matrices:

60º clockwise: $\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}$ 60º counterclockwise: $\begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}$

Time The simulation proceeds in time steps. During each time step, each critter is allowed to take one turn. These turns are taken sequentially, so each critter sees the changes to the state of the world caused by all critters that have already taken a turn during the current time step. The order in which critters take turns is fixed. Newly created critters are added to the end of the ordering. Being early in the ordering is not an advantage, however, because nothing happens to the state of the world between time steps.

2 Critter actions

The allowed actions are the following:

- **Wait.** The critter waits until the next turn without doing anything except absorbing solar energy.
- **Move forward or backward.** A critter uses some energy to move forward to the hex in front of it or backward to the hex behind it. If it attempts to move and there is a critter, food, or a rock in the destination hex, the move fails but still takes energy.
- **Turn.** It may rotate 60 degrees right or left. This takes little energy.
- **Eat.** The critter may eat some of the food that might be available on the hex ahead of it, gaining the same amount of energy as the food it consumes. When there is more food available on the hex than the critter can absorb, the remaining food is left on the hex.
- **Serve.** A critter may convert some of its own energy into food added to the hex in front of it, if that hex is either empty or already contains some food.
- **Attack.** It may attack a critter directly in front of it. The attack removes an amount of energy from the attacked critter that is determined by the size and offensive ability of the attacker and the defensive ability of the victim.
- **Tag.** The critter may tag the critter in front of it, e.g., to mark that critter as an enemy or a friend.
- **Grow.** A critter may use energy to increase its size by one unit.
- **Bud.** A critter may use a large amount of its energy to produce a new, smaller critter behind it with the same genome (possibly with some random mutations).
- **Mate.** A critter may attempt to mate with another critter in front of it. For this to be successful, the critter in front must also be facing toward it and attempting to mate in the same time step. If mating is successful, both critters use energy to create a new size-1 critter whose genome is the result of merging the genomes of its parents. Unsuccessful mating uses little energy.

3 Critter state

Critter state comprises several attributes, and also the program that drives the critter. The critter has a current location in the world and a current direction, represented as an integer between 0 and 5 as shown in Figure 1. It also has a current size and energy. The critter also has some fixed attributes: its offensive and defensive ability, the size of its memory, and the rules governing its behavior.

Two parts of critter state control how it appears to other critters: its *posture*, which it can change, and its *tag*, which other critters can change. The posture is an integer between 0 and 99, which the critter may change arbitrarily. The tag is an integer between 0 and 99 and is initially 0. A critter may use its `tag` action to change the tag of another critter, but a critter is unable to change its own tag.

Each critter has a derived attribute, its *complexity*. This is a weighted sum of the number of its rules, the size of its memory, and its offensive and defensive abilities. The energy of certain actions the critter performs increases with its complexity. The formula for complexity is found in Section 12.

4 Critter memory

Each critter has a memory called `mem`, which is an array of fixed length containing integers. The first few entries in this array have a meaning that is the same for all critter species:

- `mem[0]`: the length of the critter's memory (immutable, always at least 8)
- `mem[1]`: defensive ability (immutable, ≥ 1)
- `mem[2]`: offensive ability (immutable, ≥ 1)
- `mem[3]`: size (variable, but cannot be assigned directly, ≥ 1)
- `mem[4]`: energy (variable, but cannot be assigned directly, ≥ 1)
- `mem[5]`: pass number, explained below (variable, but cannot be assigned directly, ≥ 1).
- `mem[6]`: tag (variable, but cannot be assigned directly. Always between 0 and 99.)
- `mem[7]`: posture (assignable only to values between 0 and 99).

There are three kinds of memory entries: immutable entries that never change, variable entries that reflect the current state of the critter but that the critter's rules cannot assign to, and general-purpose mutable entries that can be both read from and assigned to by critter rules.

The size of the critter's memory must be at least 8 to accommodate these entries. If the size is larger, the remaining entries, with indices starting at 8, are general-purpose entries.

5 Rule language

The grammar for the rules is given as a context-free grammar in so-called EBNF (Extended Backus-Naur Form[1]) in Figure 3. In EBNF grammars, the right-hand side may be a regular expression. EBNF does not add any real expressive power to context-free grammars, but makes them

[1] Despite its name, EBNF is apparently not due to either Backus or Naur!

program → *rule* *rule**
rule → *condition* `-->` *command* `;`
command → *update** *update-or-action*
update-or-action → *update* | *action*
update → `mem [` *expr* `] :=` *expr*
action → `wait` | `forward` | `backward` | `left` | `right`
| `eat` | `attack` | `grow` | `bud` | `mate`
| `tag [` *expr* `]` | `serve [` *expr* `]`
condition → *conjunction* (`or` *conjunction*)*
conjunction → *relation* (`and` *relation*)*
relation → *expr* *rel* *expr* | `{` *condition* `}`
rel → `<` | `<=` | `=` | `>=` | `>` | `!=`
expr → *term* (*addop* *term*)*
term → *factor* (*mulop* *factor*)*
factor → ⟨*number*⟩ | `mem [` *expr* `]` | `(` *expr* `)` | `-` *factor* | *sensor*
sensor → `nearby [` *expr* `]` | `ahead [` *expr* `]` | `random [` *expr* `]` | `smell`
addop → `+` | `-`
mulop → `*` | `/` | `mod`

Figure 3: Grammar for critter rules

easier to express concisely. The parts of the right-hand side shown in blue are EBNF syntax and are not part of the critter language. For example, vertical bars separate alternatives, blue parentheses group symbols together, and the superscript blue asterisk is the Kleene star operator, denoting zero or more repetitions of a symbol or symbols. Terminal symbols are shown in typewriter font. Terminal symbols whose lexical representation is not fixed are shown using angle brackets, e.g., ⟨*number*⟩. Non-terminals are shown in italics, like this: *program*.

To simplify parsing, parentheses are used for grouping expressions, whereas braces are used for grouping conditions, and brackets are used for grouping arguments to commands and sensors.

The precedence of operations is specified by the grammar. All arithmetic operators associate to the left.

The parser should ignore blank lines and parts of lines that start with a double slash (`//`).

5.1 Syntactic sugar

Certain convenient abbreviations may be used in place of expressions of the form `mem[`*n*`]` for certain literal constant integers *n*. This *syntactic sugar* (so-called because it "sweetens the syntax")

may be used both when programs are read from a file, and when programs are displayed to the user. The full list of abbreviations is as follows:

Abbreviation	AST representation
`MEMSIZE`	`mem[0]`
`DEFENSE`	`mem[1]`
`OFFENSE`	`mem[2]`
`SIZE`	`mem[3]`
`ENERGY`	`mem[4]`
`PASS`	`mem[5]`
`TAG`	`mem[6]`
`POSTURE`	`mem[7]`

Therefore, a rule like `mem[3]>1000 --> mem[11] := mem[11] - mem[4];` can be written and displayed completely equivalently as `SIZE>1000 --> mem[11] := mem[11] - ENERGY;`

The syntactic sugar does *not* appear in the abstract syntax tree, however; regardless of how the expression is written in the input file or displayed to the user, the underlying abstract syntax tree node is always of the form `mem[`*n*`]`. You are encouraged to have your program display abstract syntax using the syntactic sugar, but this is not required.

5.2 Executing rules

When it is a critter's turn, it finds the first rule in its list of rules whose condition is true. It then performs all of the updates on the right-hand-side of the rule, along with the action, if any. If the command for the rule contains no action, the process repeats: starting again from the very first rule, it finds the earliest rule whose condition holds, and performs its command. This process is performed up to 999 times, after which the critter automatically performs a `wait` action. If no rule's condition is true on any pass through the rules, the critter's turn immediately ends and it performs a `wait` action on that turn.

The special memory location `mem[5]` (sugar: `PASS`) reports which pass through the rules is being done. It has the value 1 on the first pass through the rules, 2 on the second pass (if any), then 3, and so on up to a maximum of 999. It starts over again at 1 on the critter's next turn.

It may be possible to accelerate running the rules in various ways. For example, later passes might only check rules whose condition could possibly have become true. Or, if the selected rule has no effect on critter state, the critter will never select any other rule and there is no reason to run further passes. However, these sorts of optimizations are not required.

6 Sensing

A critter can sense its immediate surroundings using *sensor* expressions as described in the grammar.

- The expression `nearby[`*dir*`]` reports the contents of the hex in direction *dir*, where $0 \le dir \le 5$. Here the direction is relative to the critter's current orientation, so 0 is always immediately in front, 1 is 60 degrees to the right, and so on. (If *d* is out of bounds, its remainder when divided by 6 is used.) The contents are reported as a number *n*, as follows:
  - 0: the hex is completely empty.
  - $n > 0$: the hex contains a critter with appearance *n* (see Section 7).
  - $n < -1$: the hex contains some food, with total energy value $(-n) - 1$.
  - $n = -1$: the hex contains a rock.
- The expression `ahead[`*dist*`]` reports the contents of hex that is directly ahead of the creature at distance *dist*, using the same scheme as `nearby`. Thus, `ahead[0]` reports on the appearance of the current critter. A negative distance is treated as zero distance.
- The expression `smell` currently always evaluates to zero. This semantics will be changed later.
- The `random` expression generates a random integer from 0 up to one less than the value of the given expression. Thus, `random[2]` gives either 0 or 1 randomly. For $n < 2$, `random[`*n*`]` is always zero.

7 Critter appearance

Memory entry `mem[6]` (sugar: `TAG`) contains the critter's tag, which is initially zero and is set to some other value only when some other critter tags it. The action `tag[`*expr*`]`, where *expr* evaluates to some value *v*, causes the critter in front of it to acquire the tag *v*. The action has no effect if *v* is not a legal tag value.

The entry `mem[7]` (sugar: `POSTURE`) contains the critter's posture, which defines part of how it looks to other critters. The posture can be used as a way to signal to other nearby critters what the current critter is up to, or as a way of signaling species identity. Initially zero, the posture is set by simply assigning to its memory location.

When a critter is seen by another critter (or by itself, using `ahead[0]`), its appearance is reported as a positive integer, equal to $size * 100{,}000 + tag * 1{,}000 + posture * 10 + direction$. In other words, if the size is `SS`, the tag is `TT`, the posture is `PP`, and the direction is `D`, then the appearance is `SSTTPPD`. Note that both the critter's tag and posture are less than 100. A newly created critter has size 1, posture 0, and tag 0, so its appearance is 100,000 + *direction*. Here, *direction* is, as usual, one of $\{0, \ldots, 5\}$, and is reported relative to the observing critter's direction. For example, a direction of 3 means that the observed critter is facing in the direction opposite from that of the observing critter. Critter programs can use the operators `mod` and `/` to extract the different components of the appearance.

8 Energy and size

A critter has an initial size of 1 but can increase its size using the action `grow`. Size affects energy expenditure but also makes the critter more effective at some actions. Size also determines the maximum energy of the critter. For each point of size, the critter can hold `ENERGY_PER_SIZE` (= 500) points of energy. Any updates that would increase energy beyond this point cause excess

energy to be discarded.

If energy ever goes to (or below) zero, the critter dies. Its death adds to the food on its hex a number of food points equal to `FOOD_PER_SIZE` (= 200) points per point of critter size. Critters cannot "borrow" energy to perform an action even if the action adds enough energy to put them back in positive territory.

9 Attacking and defending

When one critter attacks another, some damage is done to the defending critter (the victim). This subtracts energy from the victim. If the victim's energy goes to zero (or lower), the victim dies and is turned into an amount of food proportional to its size.

When one critter attacks another, the damage done depends on the sizes of the two critters. If critter 1 attacks critter 2, S_1 and S_2 are the sizes of the corresponding critters, O_1 is the offensive ability of critter 1, and D_2 is the defensive ability of critter 2, the energy removed from critter 2 is:

$$\texttt{BASE\_DAMAGE} \cdot S_1 \cdot P(\texttt{DAMAGE\_INC} \cdot (S_1 \cdot O_1 - S_2 \cdot D_2))$$

where `BASE_DAMAGE` = 100, `DAMAGE_INC` = 0.2, and $P(x)$ is the *logistic function*:

$$P(x) = \frac{1}{1 + e^{-x}}$$

This formula means that critters do damage proportional to their size, but that they do only half their maximum damage if they are evenly matched against the defending critter. Damage falls off quickly to zero when attacking a critter with a higher effective defense.

10 Mutation

When a critter's genome is copied to a new critter, the copy is sometimes perfect. But with probability $p = 1/4$ there will be at least one mutation. If a mutation occurs, then there is a $1/4$ chance of further mutations. For example, an overall chance of at least two mutations is $1/16$.

A mutation is either a change to an attribute or a change to the rule set, with each equally probable. The attributes that may change are the size of the memory and the offensive and defensive abilities. A change to an attribute is an increment or decrement, chosen with equal probability, to one of these three attributes, chosen with equal probability. However, changes to attributes never reduce them below their minimal legal value (8 for memory size, 1 for offense and defense).

A mutation to the rule set is performed by randomly picking a node in the abstract syntax tree describing the entire set of rules. All nodes are chosen with equal probability. Given that a node has been selected, one of the following changes is made, with equal probability among each of the possible alternatives:

1. The node, along with its descendants, is removed. If the parent of the node being removed needs a replacement child, one of the node's children of the correct kind is randomly selected. For example, a rule node is simply removed, while a binary operation node would be replaced with either its left or its right child.

2. The order of two children of the node is switched. For example, this allows swapping the positions of two rules, or changing $a - b$ to $b - a$.
3. The node and its children are replaced with a copy of another randomly selected node of the right kind, found somewhere in the rule set. The entire AST subtree rooted at the selected node is copied.
4. The node is replaced with a randomly chosen node of the same kind (for example, replacing `attack` with `eat`, or + with *), but its children remain the same. Literal integer constants are adjusted up or down by the value of `java.lang.Integer.MAX_VALUE/r.nextInt()`, where legal, and where `r` is a `java.util.Random` object.
5. A newly created node is inserted as the parent of the mutated node. The old parent of the mutated node becomes the parent of the inserted node, and the mutated node becomes a child of the inserted node. If the inserted node has more than one child, the children that are not the original node are copies of randomly chosen nodes of the right kind from the entire rule set.
6. For nodes with a variable number of children, an additional copy of one of the children, randomly selected, is appended to the end of the list of children. This applies to the root node, where a new rule can be added, and also to command nodes, where the sequence of updates can be extended with another update.

Notice that not all mutations make sense on all node types. The mutations that may occur must result in a well-formed AST: that is, one that could be the result of parsing an input file.

11 Budding and mating

When a new critter is created by budding, it appears directly behind the critter doing the budding. When two critters mate, it appears directly behind one of the two critters, chosen at random.

When a new critter is created by budding, its rules are copied from its parent, modulo possible mutation. Its attributes are also copied from the parent, with the exception of energy, size, posture, and tag. Energy is set to a constant `INITIAL_ENERGY` (= 250), size is always set to 1, posture is always set to 0, and the tag is always set to 0. All memory locations at or above index 8 are set to zero in the newly created critter.

When two critters mate, however, they exchange genetic material to form the new critter. Attributes 0–2 are chosen at random from one of the two critters. Attributes 3–5 are chosen as for budding. The new rule sequence is chosen by picking the corresponding rule in sequence from either the 'mother' or the 'father', at random. Thus, the new critter inherits, in general, some rules from each parent. If the mother or father have different-sized rule sets, the new rule set either has the size of the mother or the father, randomly chosen. Thus, if the mother and father have identical genomes, and there are no mutations, the child will have the same genome too.

12 Energy

Different actions take different amounts of energy, even waiting for a turn. The energy cost of different actions is as follows:

- `wait`: this action *increases* the critter's energy by its own size times `SOLAR_FLUX` (= 1).

- `tag`, `left` (turn), `right` (turn), and `eat`: energy equal to the critter's size.
- `forward` and `backward`: energy equal to the critter's size times `MOVE_COST` (= 3).
- `serve`: the amount of energy served onto the hex in front, plus the creature's size. A critter can use the action `serve` to send its own energy down to zero, killing it, but not below. A critter that kills itself in this way deposits additional food onto the hex in the usual way.
- `attack`: energy equal to the critter's size times `ATTACK_COST` (= 5).
- `grow`: energy equal to *size* · *complexity* · `GROW_COST` (= 1).
- `bud`: `BUD_COST` (= 9) · *complexity* energy.
- `mate`: `MATE_COST` (= 5) · *complexity* energy.

Several of these energy costs depend on the critter complexity. If r is the number of rules in the critter program and *offense* and *defense* are the critter's offensive and defensive abilities, the critter complexity is equal to:

$$r \cdot \texttt{RULE\_COST} + (\textit{offense} + \textit{defense}) \cdot \texttt{ABILITY\_COST}$$

Most actions take the same energy whether they are successful or unsuccessful. One exception is the `mate` action, which only costs as much as `turn` if it is unsuccessful.

13 Handling out-of-bounds arguments

One important principle is that syntactically legal critter programs always evaluate successfully. Even when an argument to a sensor or action might seem to be out of bounds, the expression or action will complete. Mutation to critter programs can never cause the simulation as a whole to fail.

Ostensibly out-of-bounds arguments are handled in the following way for the various language constructs:

- `mem[`*expr*`]`: A read from a memory location *expr* where *expr* is not a valid memory index always returns 0. An update to an illegal memory location has no effect. An update to a memory location whose value is constrained (e.g., `mem[7]`) also has no effect if the value is out of bounds for that location.
- `tag[`*expr*`]`: If *expr* evaluates to a tag value that is illegal, it has no effect.
- `+` and `-`: these operate exactly like Java + and -.
- `/` and `mod`: If the divisor is zero, the result of the expression is also zero.
- `nearby[`*expr*`]`: the nonnegative remainder of *expr* when divided by 6 is used as the direction.
- `random[`*expr*`]`: always zero when *expr* < 2.

14 Example critter program

The following critter program should be able to survive, find food, and reproduce.

```
POSTURE != 17 --> POSTURE := 17; // we are species 17!
nearby[3] = 0 and ENERGY > 2500 --> bud;
{ENERGY > SIZE * 400 and SIZE < 7} --> grow;
ahead[0] < -1 and ENERGY < 500 * SIZE --> eat;
// next line attacks only other species
(ahead[1] / 10 mod 100) != 17 and ahead[1] > 0 --> attack;
ahead[1] < -5 --> forward;
ahead[2] < -10 and ahead[1] = 0 --> forward;
ahead[3] < -15 and ahead[1] = 0 --> forward;
ahead[4] < -20 and ahead[1] = 0 --> forward;
nearby[0] > 0 and nearby[3] = 0 --> backward;
// karma action: donate food if we are too full or large enough
ahead[1] < -1 and { ENERGY > 2500 or SIZE > 7 } --> serve[ENERGY / 42];
random[3] = 1 --> left;
1 = 1 --> wait; // mostly soak up the rays
```

15 Constants

Numbers used in this document are mostly parameters that have symbolic names. We may fiddle with the values of these parameters to make the simulation more interesting, so you should always use the symbolic names rather than hard-coding them into your program. We are providing a file containing the current values of these constants; your program should parse the file at run time to set them to the correct values.

The current values of the simulation constants are shown in Figure 4. Most of the constants are integers, but a few are real numbers, as indicated by the presence of a decimal point.

16 Challenges

This project has several different kinds of subsystems. One of the major challenges will be to keep the different parts of the system separate, so that, for example, your simulation code is entirely separate from your graphics code. This particular separation will be crucial for making a distributed version of the program, since the simulation and the displays will be done on a completely different machines. Therefore the simulation code needs to avoid knowing about or naming the graphics code. Similarly, you will want to separate different parts of the simulation into different modules. Your programming tasks will be simpler if the interpretation of critter rules is kept separate from the mechanics of the world and even of the critter itself.

Thoughtful up-front design with your partner will save you a tremendous amount of time later on. Meet early with your partner and decide on how you will structure your project and agree on interfaces and specifications that connect the different parts of the code.

Name	Value	Description
BASE_DAMAGE	100	The multiplier for all damage done by attacking
DAMAGE_INC	0.2	Controls how quickly increased offensive or defensive ability affects damage
ENERGY_PER_SIZE	500	How much energy a critter can have per point of size
FOOD_PER_SIZE	200	How much food is created per point of size when a critter dies
MAX_SMELL_DISTANCE	10	Maximum distance at which food can be sensed
ROCK_VALUE	-1	The value reported when a rock is sensed
COLUMNS	50	Default number of columns in the world map
ROWS	68	Default number of rows in the world map
MAX_RULES_PER_TURN	999	The maximum number of rules that can be run per critter turn
SOLAR_FLUX	1	Energy gained from sun by doing nothing
MOVE_COST	3	Energy cost of moving (per unit size)
ATTACK_COST	5	Energy cost of attacking (per unit size)
GROW_COST	1	Energy cost of growing (per size and complexity)
BUD_COST	9	Energy cost of budding (per unit complexity)
MATE_COST	5	Energy cost of successful mating (per unit complexity)
RULE_COST	2	Complexity cost of having a rule
ABILITY_COST	25	Complexity cost of having an ability point
INITIAL_ENERGY	250	Energy of a newly birthed critter
MIN_MEMORY	8	Minimum number of memory entries in a critter

Figure 4: Constants

17 Extensions

Adding extensions to the critter simulation may be done for **karma**, but you need not. Chances are you'll have your hands full just implementing the project as is, so be judicious about adding new features.

Possible extensions might be additions to the critter language (abbreviations? function definitions?), to the critter model (better sensory capabilities?), changes to the world (volcanos? water? plants that grow food? climate gradients?), better user control over the world view (zooming and panning? high-level critter commands?). Feel free to be creative. If your extensions might interfere with our testing, for example by making our critter programs invalid, be sure to support a command-line flag `-compatible` that turns off your extra features. We recommend being backward compatible to our specification in any case.

18 Tournament

We will have a tournament at the end of the semester when you can bring in some critter programs to compete in various events such as survival, food gathering, and a maze race. We encourage participation in the tournament, and there will be free food, but the fun is optional. We will post more information about the tournament as it approaches.