CS2112—Fall 2012

Final Project

Simulating Evolving Artificial Life

Version of November 30, 2012

The final programming assignments for this course make up a final project that you will work on as part of a two-person team. In this project, you will simulate a simple world of animals 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.

The animals, called critters, are located on a regular, hexagonal grid. Plant-like food grows on the surface of the world. Critters must find and eat it to stay alive. Critters may also attack each other; when critters run out of energy, they die and become food. If they accumulate enough energy, they can reproduce.

The genome of a critter is actually an event-driven program that determines what it does on a given time step. When critters reproduce, the genome is copied from the parent or parents to the new critter, with possibly some mutations applied. 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.

- The simplified grammar was simplified more to make it clear how to parse terms and factors non-recursively. (11/9)
- Typo in spec for `ahead` corrected. (11/6)
- Initial appearance corrected in Section 11. (10/30)
- Mutation 5 was clarified. (10/20)
- The grammar was simplified slightly. The *term* nonterminal now generates a term rather than terms, the *factor* nonterminal generates a factor rather than factors, and the atom nonterminal was removed. This does not affect the critter language, and it should not affect your mutation code if you've done it right. (10/18)

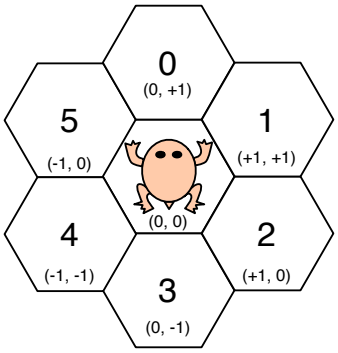

Figure 1: Tiles and directions

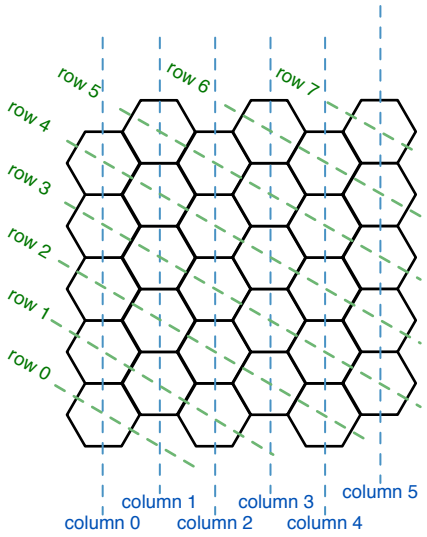

Figure 2: Rows and columns of hexes

- The new mutation type was extended to apply to updates. (10/18)
- One additional mutation type was added, to take effect in Assignment 6. (10/17)
- The behavior of various language constructs on out-of-bounds values was summarized in Section 14. (10/17)
- The result of `ahead` on negative arguments was changed. (10/17)
- The result of a copying mutation was clarified. (10/7)

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

Hexes are either rock hexes or hexes that contain other things. Rocks are inert obstacles that critters cannot move onto or over. A non-rock hex may be empty, or may contain any or all of the following: a critter, a plant, or some food. A hex cannot contain more than one critter or more than one plant.

Food is created on a hex when a critter dies there. Other critters may then eat the food to gain energy. Plant hexes can also provide food to critters. When a critter eats the plant on its hex, the plant is destroyed and the critter gains a fixed amount of energy (`ENERGYPERPLANT` = 100).

Plants can grow. Every hex containing a plant will, with a small probability on each turn and for each of its neighboring hexes that does not already contain a plant, generate a new plant. Note that a plant may generate a plant on more than one adjacent hex on a given turn, but generation of

plants on its adjacent hexes occurs independently. In addition, a few plants spontaneously appear at random on non-rock hexes on each turn.

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 row and column coordinates of various hexes. The world has a fixed, roughly rectangular shape, in which each column contains the same number of hexes. 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 $\texttt{MAX\_COLUMN} = 5$ and $\texttt{MAX\_ROW} = 7$. A coordinate such that $c \not\leq \texttt{MAX\_COLUMN}$ lies off the east edge of the world. A coordinate such as (1,0), where $c \not\leq 2r$, lies off the south edge of the world. And a coordinate where $c + (2 \cdot \texttt{MAX\_ROW} - \texttt{MAX\_COLUMN}) \not\geq 2r$ lies off the north edge of the world.

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. Using slices, we can compactly represent the length of the shortest path between two hexes:

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

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 updates cause by all critters that have taken a turn during the current time step. The order in which critters take turn is fixed. Newly created critters are added to the end of the order.

Being early in the ordering is not a significant advantage because other behavior of the simulation occurs throughout a given time step. In particular, plant growth occurs randomly before each critter's turn rather than all at once at the beginning of the time step. However, since simulated time is measured in time steps rather than in critter turns, the probability that plants grow on a given critter turn must be scaled down to compensate for the number of critters taking turns.

2 Critter actions

The allowed actions are the following:

- **Wait.** The critter waits until the next turn without doing anything.
- **Move forward or backward.** A critter may 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 or 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.** A critter may eat all food that is sitting on its current hex. It gains energy from any food that it eats.
- **Attack.** It may attack the critter in front of it, assuming there is one. 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—for example, indicating that the critter is a 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 with the same genome (possibly with some 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 new size 1 critter containing a merge of their genomes.

3 Critter state

Critter state consists of several attributes in addition to the program that drives the critter. The critter has a current location in the world and a current direction, represented as an integer in 0..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 999, which the critter may change arbitrarily. The tag is an integer between 0 and 999 and is initially 0. It may be changed by any other critter using its `tag` action, but a critter is not able 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 depends on its complexity. The formula for complexity is given on page 10.

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 9)
- `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, always positive)
- `mem[5]`: rule counter, explanation below (variable, but cannot be assigned directly).
- `mem[6]`: event log (variable, but cannot be assigned directly)
- `mem[7]`: tag (variable, but cannot be assigned directly. Always between 0 and 999.)
- `mem[8]`: posture (assignable but only to values between 0 and 999).

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 9 to accommodate these entries. If the size is larger, the remaining entries, indexed starting at 9, 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 easier to express concisely. Terminal symbols are shown in typewriter font. Terminal symbols whose lexical representation is not fixed are shown using angle brackets: for example, $\langle number \rangle$. Non-terminals are shown in italics, like this: *program*.

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 1000 times, after which the critter automatically performs a `wait` action. If on any pass through the rules, no rule's condition is true, the critter's turn ends and it performs a `wait` action.

The special memory location `mem[5]` reports which pass through the rules is being done. It has the value 0 on the first pass through the rules, 1 on the second pass (if any), then 2, and so on. It starts over again at 0 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.

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

$$
\begin{array}{rcl}
\textit{program} & \rightarrow & \textit{rule}^* \\
\textit{rule} & \rightarrow & \textit{condition}\ \texttt{-->}\ \textit{command}\ \texttt{;} \\
\textit{command} & \rightarrow & \textit{update}^*\ \ \textit{update-or-action} \\
\textit{update-or-action} & \rightarrow & \textit{update}\ \mid\ \textit{action} \\
\textit{update} & \rightarrow & \texttt{mem [}\ \textit{expr}\ \texttt{] :=}\ \textit{expr} \\
\textit{action} & \rightarrow & \texttt{wait} \mid \texttt{forward} \mid \texttt{backward} \mid \texttt{left} \mid \texttt{right} \\
 & \mid & \texttt{eat} \mid \texttt{attack} \mid \texttt{grow} \mid \texttt{bud} \mid \texttt{mate} \mid \texttt{tag [}\ \textit{expr}\ \texttt{]} \\
\textit{condition} & \rightarrow & \textit{conjunction}\ (\ \texttt{or}\ \textit{conjunction}\ )^* \\
\textit{conjunction} & \rightarrow & \textit{relation}\ (\ \texttt{and}\ \textit{relation}\ )^* \\
\textit{relation} & \rightarrow & \textit{expr}\ \textit{rel}\ \textit{expr}\ \mid\ \texttt{\{}\ \textit{condition}\ \texttt{\}} \\
\textit{rel} & \rightarrow & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{!=} \\
\textit{expr} & \rightarrow & \textit{term}\ (\ \textit{addop}\ \textit{term}\ )^* \\
\textit{term} & \rightarrow & \textit{factor}\ (\ \textit{mulop}\ \textit{factor}\ )^* \\
\textit{factor} & \rightarrow & \langle number \rangle \mid \texttt{mem [}\ \textit{expr}\ \texttt{]} \mid \texttt{(}\ \textit{expr}\ \texttt{)} \mid \textit{sensor} \\
\textit{sensor} & \rightarrow & \texttt{nearby [}\ \textit{expr}\ \texttt{]} \mid \texttt{ahead [}\ \textit{expr}\ \texttt{]} \mid \texttt{random [}\ \textit{expr}\ \texttt{]} \\
\textit{addop} & \rightarrow & \texttt{+} \mid \texttt{-} \\
\textit{mulop} & \rightarrow & \texttt{*} \mid \texttt{/} \mid \texttt{mod}
\end{array}
$$

Figure 3: Grammar for critter rules

- The expression `nearby[`dir`]` reports the contents of the hex in direction dir, where $0 \leq dir \leq 5$. 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 no critter, and total food value is $-n$ (including the plant on the hex, if any).
  - $n = -1$: 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`. If $dist$ is not positive, `ahead` evaluates as if it were `ahead[`$-dist$`]`, but any critter that is on the hex is ignored. Thus, `ahead[0]` reports on the hex under the critter, ignoring the critter itself.
- 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`]` always is zero.

7 Critter appearance

The entry `mem[7]` 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[8]` 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. Initially zero, the posture it 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 $tag * 100{,}000 + size * 1{,}000 + posture$. Note that both the critter's tag and posture are less than 1,000. A newly created critter has size 1, posture 0, and tag 0, so its appearance is 1,000.

8 Energy and size

A critter has an initial size of 1 but can increase this through the `grow` action. Size affects energy expenditure but also the critter's effectiveness 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. Attempts to increase energy beyond this point result in excess energy being 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 size.

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, and 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 $\texttt{BASE\_DAMAGE} = 100$, $\texttt{DAMAGE\_INC} = 0.2$, 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 may be perfect. But with probability $p = 1/4$ there will be at least one mutation. If there is one mutation, there is then a 1/4 chance (i.e. 1/16 overall) of a second mutation, and so on for possible additional mutations.

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 (9 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 is removed. If its parent node needs a replacement node, one of its children of the right kind is used. The child to be used is randomly selected. Thus, rule nodes are simply removed, but binary operation nodes would be replaced with either their left or 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 previous parent 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 whole rule set.

6. For nodes with a variable number of children: an additional copy of one of the children 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. (This does not have to be implemented for Assignment 4.)

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 9 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 from between the two critters randomly. 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`, `turn`, `eat` : energy equal to the critter's size.
- `forward` and `backward`: energy equal to the critter's size times `MOVE_COST` where `MOVE_COST` = 3.
- `eat` : the same energy as `wait`.
- `attack` : energy equal to the critter's size times `ATTACK_COST` = 5.

- `tag` : the same energy as `wait`.
- `grow` : energy equal to $size \cdot complexity \cdot$ `GROW_COST` where `GROW_COST` $= 1$.
- `bud` : `BUD_COST` $\cdot$ $complexity$ energy, where `BUD_COST` = 9.
- `mate` : `MATE_COST` $\cdot complexity$ energy, where `MATE_COST` = 5.

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:

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

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

13 Events

The entry `mem[6]` is the event log. It records what happened to the critter in the last time step, as described below. A single event is encoded as an integer between 0 and 999, as captured in the following table. As usual, directions d are relative to the critter's current orientation. Damage k is expressed as a percentage (0–99) of the total energy of the attacked critter.

Event	Encoding
This critter was attacked from direction d	100 + d
This critter was tagged from direction d	200 + d
This critter did k% damage to the critter it attacked	300 + k

Up to three events can be encoded into `mem[6]`. At the end of a critter's turn, its event log is set to zero. Each time an event happens, encoded as value `v` according to the above table, its event log is updated as follows:

```
mem[6] := (mem[6] % 1000000) * 1000 + v
```

In other words, the previous events recorded in that turn, if any, shift left by three digits. If more than three events occurred, some will be forgotten.

For example, if on the last turn the critter did 30 damage to the critter in front of it, and that critter had 100 energy, and then subsequently the critter was attacked from behind, its event log would contain 330,103.

14 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.

Handling ostensibly out-of-bounds arguments is 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 (in particular, `mem[8]`) 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 remainder of *expr* when divided by 6 is used as the direction.
- `random[`*expr*`]`: always zero when $expr < 2$.

15 Example critter program

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

```
mem[8] != 17 --> mem[8] := 17;
nearby[3] = 0 or nearby[3] < 0-1 and mem[4] > 2500 --> bud;
mem[4] > mem[3] * 400 and mem[3] < 7 --> grow;
ahead[0] < 0-1 and mem[4] < 500 * mem[3] --> eat;
ahead[1] != 17 and ahead[1] > 0 --> attack;        // attack other species
ahead[1] < 0-5 --> forward;
ahead[2] < 0-10 and ahead[1] = 0 --> forward;
ahead[3] < 0-15 and ahead[1] = 0 --> forward;
ahead[4] < 0-20 and ahead[1] = 0 --> forward;
nearby[0] > 0 and nearby[3] = 0 or nearby[3] < 0-1 --> backward;
1=1 --> left;
```

16 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.

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_PLANT	100	How much energy is obtained by eating a plant
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	The maximum distance at which food can be sensed
ROCK_VALUE	-1	The value reported when a rock is sensed
MAX_COLUMN	99	The maximum column index in the world map
MAX_ROW	139	The maximum row index in the world map
MAX_RULES_PER_TURN	1000	The maximum number of rules that can be run per critter turn
PLANTS_CREATED_PER_TURN	2	How many plants are created from nothing on each simulation time step
PLANT_GROW_PROB	0.005	The probability of a plant sprouting an adjacent plant on a given simulation time step
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	9	Minimum number of memory entries in a critter
MAX_SMELL_DIST	10	Maximum distance at which food can be smelled

Figure 4: Constants

17 Challenges

This project has several very 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 an 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 design up front along 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.

18 Extensions

You may add extensions to the critter simulation, but you need not. 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? 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.