Bugs! Programming Competition Instructions

Welcome to the Bugs! Programming Competition!

In this programming competition, your job is to program an artificial species of ant using our specially-built programming language, called “Bugs!” Once you program your ant species, you’ll then be able to have it compete against other students' species within a virtual world to see which ant colony is the most prolific.

During each round of the competition, four different ant species (each species written by a different student or team of students) will compete against each other. At the start of the round, each species’ anthill will produce five initial ants for the colony. These ants must then forage for food and bring it back to the anthill so the queen ant in the anthill can produce more ants. The ant species that produces the most total ants (at least 6 total) by the end of the round wins that round. If two ant species produce the same number of ants, then the colony that did so first wins that round. The winning species will then advance to the next round, competing against other winning species, until there is a single victorious species and the competition is over.

To win a round of the competition, each of your ants must forage for food, pick it up, bring it back to your anthill, and drop it there so your queen can eat it and have enough energy to produce more ants. The more food your ants bring back to the anthill, the more baby ants your queen ant can produce, and the more likely you are to win the round.

But beware, in addition to enemy ants from other contestants, there are also other dangers within the 64x64 square virtual field. Evil grasshoppers will devour food up and may bite your ants. There are also patches of poison that have been spread around the virtual field by mean humans; it’ll cause nasty damage to your ants. And there are pools of water which when stepped on, will slow an ant down. And finally, pebbles block your ants’ movement, making it more difficult for them to find their anthill.

In addition, as each second of the simulation passes, each ant burns a little of its energy, and so it needs to occasionally eat some food to stay healthy. If an ant doesn’t eat any food, it’ll eventually die of starvation. Similarly, if an ant is bitten by an enemy ant or a grasshopper, it will lose lots of energy. Therefore, each ant will need to occasionally eat some food to restore its health when it’s bitten.

Like real ants, your ants have the ability to emit pheromones and then smell these pheromones at a later time. Just like bread crumbs dropped by Hansel and Gretel, pheromones are temporary scent markers that can be used to help your ants find their way back home. But beware, pheromones disappear over time. Each ant colony has its own pheromone scent, and your colony’s ants can only smell pheromones emitted by its own ants. If any ant in your colony

releases a pheromone scent on a square, then all of its brothers and sisters can smell that pheromone scent on the square as well.

So what can an ant do? It can:

- Move forward
- Adjust the direction that it's facing
- Pick up food that is on its same square
- Drop food that it's holding, for instance, on top of the ant's anthill
- Eat food that it's holding to gain energy/health
- Bite an enemy insect that's standing on the same square, to damage the enemy's health
- Emit a pheromone in the current square
- Smell if there's a pheromone or an danger (e.g., poison or an enemy) in the square directly in front of the ant
- Smell if it's standing on the same square as food, its anthill, or an enemy
- Determine if it was just just blocked from moving forward, for instance because a pebble was in the way

You program your ant by using different combinations of the above behaviors to construct its artificial brain. You'll place your program's instructions in a text file and name it. For example, you might name your ant's program *janes-ant.bug* or *larrynguyen.bug*. All ant programs must have a .bug file extension, just like Java source files have a .java extension, and C++ source files have a .cpp extension.

Below is an example of a Bugs! Program, which might be stored in a file called *dumbant.bug*. It has two parts:

1. The very **first line** specifies the name of your ant colony, for example "DumbAnt". Your ant colony's name must have 8 or fewer letters in its name (e.g., CareyAnt, Shmoopy, etc.).
2. The **remaining lines** are the actual programming instructions that control the ant's brain

```
colony: DumbAnt

// this program controls a single ant and causes it to move
// around the field and do things.
// this ant moves around randomly, picks up food if it
// happens to stumble upon it, eats when it gets hungry,
// and will drop food on its anthill if it happens to be
// stumble back on its anthill while holding food.

// here are the ant's programming instructions, written
// in our "Bugs!" language
```

```
start:
      faceRandomDirection  // face some random direction
      moveForward      // move forward
      if i_am_standing_on_food then goto on_food
      if i_am_hungry then goto eat_food
      if i_am_standing_on_my_anthill then goto on_hill
      goto start       // jump back to the "start:" line

on_food:
      pickUpFood
      goto start       // jump back to the "start:" line

eat_food:
      eatFood          // assumes we have food - I hope we do!
      goto start       // jump back to the "start:" line

on_hill:
      dropFood         // feed the anthill's queen ant so she
                       // can produce more ants for the colony
      goto start       // jump back to the "start:" line
```

As you can see above, each ant program is comprised of a set of simple instructions like faceRandomDirection (to make the ant face a new, random direction), moveForward (to make the ant move one square forward), pickupFood (to pick up food from the current square if there is food there), eatFood (to eat food if it's being carried by the ant), dropFood (to drop food that the ant was carrying on its current square). There must only be one such command per line in the Bugs! programming language.

Each ant his its own ant brain and is provided with its own copy of your Bugs! Program. So each ant's program operates entirely on its own.

Each ant executes one instruction after another, following your Bugs! program from top to bottom. So, in the ant program above, when a new ant is born, the ant will start by executing the first instruction and it will face a random direction (faceRandomDirection). Then the ant's brain will advance to the next instruction, and will attempt to move forward (moveForward). Then the ant will advance to the next instruction, and check to see if it is standing on food, and so on.

You can see that the program has many comments, which are started by two forward slashes:

```
      // I just picked up some food! Now I can eat it!
```

These comments can help you, the programmer, keep track of details that you might otherwise forget. Your programs may have as many comments as you like.

A line of your Bugs! program might have a single word, followed by a **colon**, e.g.:

```
Start:
```

or

```
on_hill:
```

This is called a **label**. A label is just a name that identifies a particular line of your program. Your programming instructions may then reference such a label with a *goto* command or an *if statement* (described below). For example, this line**:**

```
goto start      // jump back to the "start:" line
```

will cause the ant's brain to immediately jump to the line of the program labeled **start:** The ant will then continue executing instructions from that line onward.

As you can see, the ant can examine its surroundings and determine its own internal "state of being" using if statements.

```
if i_am_standing_on_food then goto on_food
if i_am_standing_on_my_anthill then goto on_hill
```

Each if statement may check a single condition, e.g., ***i_am_standing_on_food***, and if that condition is met (true), then the if statement will transfer control to the line of the Bugs! program with the specified "label:". For example, the following code will cause the ant to pick up food if it's standing on the same square as the food:

```
if i_am_standing_on_food then goto do_something_when_on_food
faceRandomDirection
moveForward
... // other instructions
... // other instructions
... // other instructions

do_something_when_on_food:  // this is a label
    pickUpFood // I just picked up some food! Now I can eat it!
    eatFood    // Nom nom nom
```

If an if statement's condition is not met (e.g., the ant is not standing on food), then the if statement has no effect and the ant's brain simply advances to the next instruction. In this

example, that would result in the ant facing a new random direction and then attempting to move forward.

Command Reference

Here is a list of all the commands you can use:

bite

If an ant is on the same square as either an enemy ant (from another colony/species) or a grasshopper (either a baby or adult grasshopper), then this command will cause your ant to bite the enemy. If there are multiple enemies on the same square as your ant, then this command picks one of those enemies randomly and causes your ant to bite them. Biting the enemy does damage to the enemy, lowering its health and possibly killing it. If your ant is bitten, it can restore its health by eating food.

dropFood

An ant that is carrying food in its mouth can drop that food. Using the dropFood command drops all of the food currently carried by the ant. Typically, an ant will drop food on top of its anthill, so that it can feed the queen ant. She can then produce more baby ants.

eatFood

Allows your ant to eat a unit of food, assuming your ant actually is holding food. Your ant needs to eat food or it will eventually starve. An ant can't eat food that is just sitting on the ground; it can only eat food that it's already picked up in its jaws. An ant can eat multiple times if it desires, to increase its overall health. However, every bit that an ant eats takes away from food that could otherwise be provided to the anthill, so be careful.

emitPheromone

This allows the ant to drop a pheromone scent on its current square. This pheromone will dissipate over time and eventually disappear altogether. A pheromone dropped by any ant in your colony can be smelled by all other ants in your colony, but not by ants from other colonies.

faceRandomDirection

This command causes your ant to face a random direction (e.g., up, down, left or right).

goto *someLineOfYourProgram*

Your ant can use the goto command to jump to a different line of your program. This command, when executed, will immediately transfer control to the specified line. For the goto command to work, your program must have a line with the specified label:

```
…       // other instructions

goto someLineOfYourProgram

…       // other instructions

someLineOfYourProgram:     // this line must be somewhere in your program
…       // do something useful
```

generateRandomNumber *someNumber*

An ant can generate a random number in its brain, between 0 and the specified numeric value minus 1. For example, to generate a number between 0 and 10, the ant would use this command:

```
generateRandomNumber 11
```

Or to generate a random number between 0 and 99, the ant could do this:

```
generateRandomNumber 100
```

Once an ant generates a random number, it can use this value to alter its behavior. See the *if statement* section below for more details on how to do so.

moveForward

This command causes your ant to move forward one square in the direction it's currently facing. If that direction is blocked by a pebble, then the ant will not be able to move forward.

pickUpFood

If your ant is standing on the same square as some food, it can pick this food up into its jaws. An ant can pick up more than one unit of food at a time if it likes.

rotateClockwise and **rotateCounterClockwise**

These commands can be used to rotate the direction an ant is facing either 90° clockwise or 90° counterclockwise.

If Statements

Your ant can check its current state as well as the state of the virtual field environment by using if statements. All if statements have the following format:

if *someCondition* then goto *someLabel*

Where *someCondition* is one of the following conditions:

i_smell_danger_in_front_of_me
i_smell_pheromone_in_front_of_me
i_was_bit
i_am_carrying_food
i_am_hungry
i_am_standing_on_my_anthill
i_am_standing_on_food
i_am_standing_with_an_enemy
i_was_blocked_from_moving
last_random_number_was_zero

For example, the statement:

if **i_smell_danger_in_front_of_me** then goto changeDirection

will check if the ant smells either an enemy ant (from a different colony), a grasshopper or poison in the square directly in front of it. If so, it will cause the ant's program to jump to the specified label:

```
changeDirection:
        faceNewRandomDirection     // change my direction to a new random one
        moveForward                // run, run, run!
```

Here are details on the full list of valid if conditions:

if **i_smell_danger_in_front_of_me** then goto *someLabel*

Checks if an ant smells either an enemy ant, a grasshopper, or poison in the square in front of it. If so, the program will goto the specified label.

if **I_smell_pheromone_in_front_of_me** then goto *someLabel*

Checks if an ant smells a pheromone released by itself or another ant in its colony, in the square directly in front of it. If so, the program will goto the specified label. Ants cannot smell the pheromones emitted by ants of other colonies.

if **I_was_bit** then goto *someLabel*

Checks if an ant was recently bit by an enemy ant/grasshopper while on the current square. If so, the program will goto the specified label.

if **i_am_carrying_food** then goto *someLabel*

Checks if an ant is carrying food. If so, the the program will goto the specified label.

if **i_am_hungry** then goto *someLabel*

Checks if an ant is hungry (i.e., it needs to eat food quickly to replenish its energy, or it will die). If so, the program will goto the specified label.

if **I_am_standing_on_my_anthill** then goto *someLabel*

Checks if an ant is standing on the same square as its anthill. If so, the program will goto the specified label.

if **i_am_standing_on_food** then goto *someLabel*

Checks if an ant is standing on the same square as food, which can be picked up. If so, the program will goto the specified label.

if **i_am_standing_with_an_enemy** then goto *someLabel*

Checks if an ant is standing on the same square as an enemy ant (from a different colony) or a grasshopper. If so, the program will goto the specified label.

if **i_was_blocked_from_moving** then goto *someLabel*

Checks if an ant was just blocked from moving (e.g., by a pebble that was in the way). If so, the program will goto the specified label.

if **last_random_number_was_zero** then goto *someLabel*

Checks if the last random number that was generated by the **generateRandomNumber** command is equal to zero. If so, the program will goto the specified label.

This can be used to cause an ant to exhibit interesting, random behaviors. For example, the program below will cause the ant to continue to move in the same direction until the ant happens to generate a random value of zero. There's a one in ten chance of the ant doing so (with a command of generateRandomNumber 10), so an ant that uses this approach will

generally walk an average of ten steps in the same direction before switching directions randomly and walking in a new direction.

```
moveAnt:
      generateRandomNumber 10
      if last_random_number_was_zero then goto changeDirection

      moveForward
      goto moveAnt

changeDirection:
      faceRandomDirection
      moveForward
      goto moveAnt
```

The Field Data File

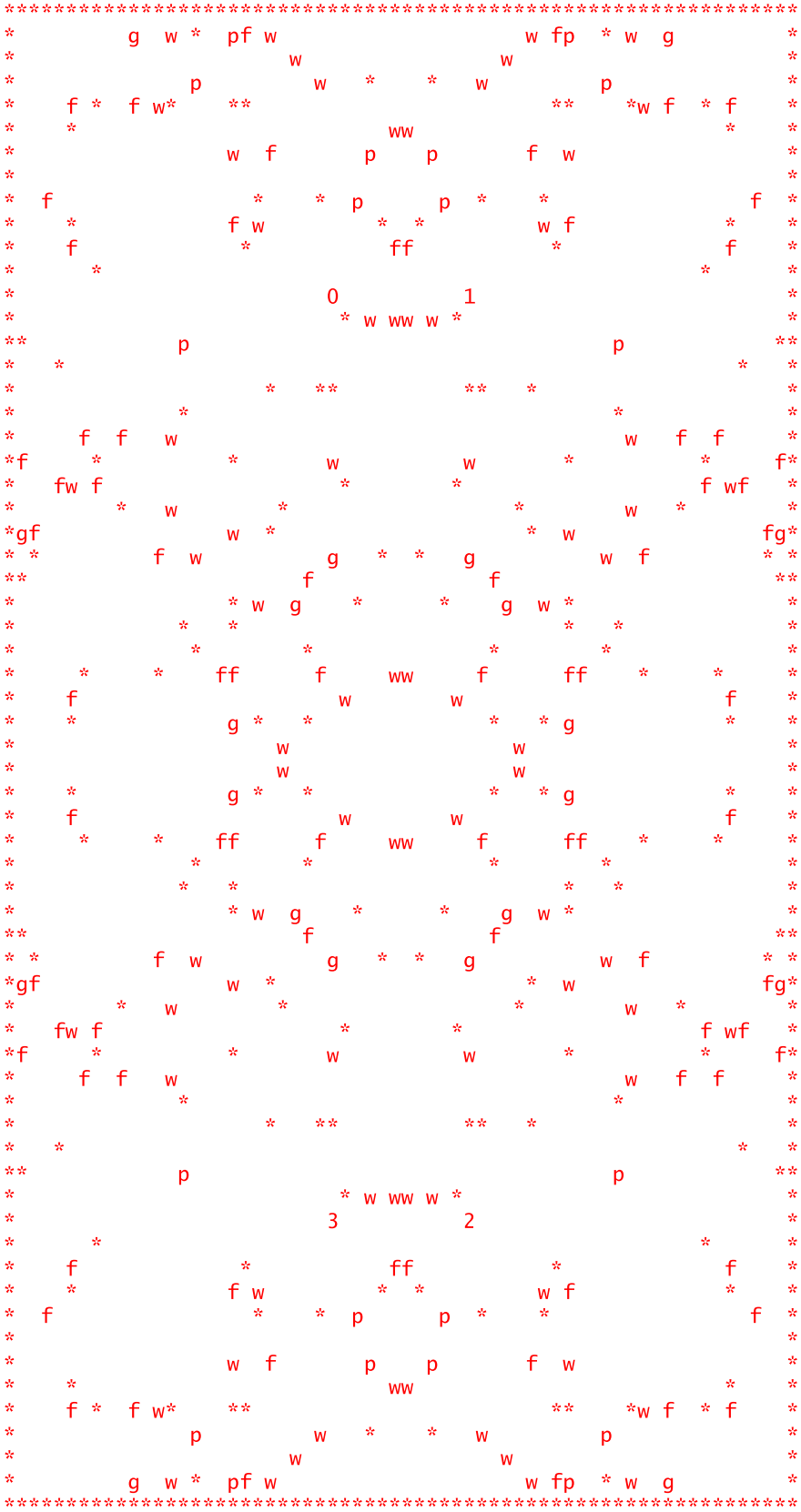
You can test your ant program in multiple different virtual fields, each with different amounts of grasshoppers, poison, pools of water, pebbles, etc. Feel free to create many different field data files as you like to test your ant's logic.

Each field data file is a simple text file. (You can edit it with Notepad on Windows, or TextEdit on a Mac. Use a fixed-width font so the columns appear properly aligned when you're editing.)

The field data file must be exactly 64 characters wide, by 64 characters high. The top and bottom rows and the left and right columns of the field must contain a pebble at each position. See the example below.

The '*' characters designate pebbles which block movement of the ants, 'g' characters specifies the starting locations of baby grasshoppers, '0', '1', '2' and '3' specify the location of the four colonies' anthills, 'w' characters specify pools of water, the 'f' characters specify piles of food, and the 'p' characters specify the locations of poison.

Each field must contain at least one anthill, designated by a 0. If you want a field to have more than one anthill, you can add a 1, 2, and 3 to the data file as well.



Trying out Your Ants - Running The Competition

You can run the competition program like this from the (Windows/Mac) command shell like this:

BUGS.EXE field.txt sillyant.bug killer.bug silly.bug anthrax.bug

where the BUGS.EXE file is the competition program. The next item is the filename of the data file that holds the name of the field data file that contains the layout of the virtual world where your ants will compete, and the following one to four filenames are the Bugs programs that will be used to control the ants in the different colonies in the competition. For example, sillyant.bug would contain the programming logic for anthill #0, killer.bug contains the programming logic for anthill #1, and so on. The field and ant data files must be in the same folder as your BUGS.EXE executable so it can find and load them, or you may specify a full pathname to these files if you like.

If you don't know how to use the Windows or Mac command shell, ask your Computer Science teacher for help.

Hints

1. **Build your ant incrementally:** Don't try to build the perfect, complex ant all at once. Instead, build simple Bugs! programs (10-20 lines) that exercise just a few simple features and see how they work. Then make your complete ant from the little pieces, once you know the pieces work properly.
2. **Don't forget to check if your ants are hungry and, if so, eat:** Make sure your ants eat or they'll die of starvation (or from being bitten by enemies). Ants can only eat once they've already picked up and are holding food in their jaws. So an ant can't just eat food if it's on the same square as the food.
3. **Use pheromones to help your ants navigate back to their anthill:** You can only win the competition if your ants actually bring food back to your anthill to feed the queen ant, so she produces more ants. Your ants can emit pheromones to help them find their way back to their anthill. Be creative! Use Google search to find out how real ants use pheromones to navigate, or perhaps come up with your own creative method.
4. **Try different fields:** Your ant program will be tested in a number of different environments (some with more poison than others, some with more pebbles than others, etc.). So try creating many different fields and see how your ants do in each type of environment. Then optimize your ants so they work well in many different environments.
5. **Make frequent backups:** Once you have an ant working, make a backup copy of its program so you have it just in case you introduce a bug (pun intended) into your program. Always have a backup handy so you have something to enter into the competition.

6. **Try to have fun:** Remember, this competition is meant to be fun, so try to work well with other people, try to learn new things, and enjoy yourself! Programming is as much about working together as it is about solving hard problems, so use this as an opportunity to learn to work well with others.

GOOD LUCK!