Battle of the Ants!

Welcome to the Battle of the Ants Programming Contest!

In this contest, your job is to come up with a set of rules to be followed by a colony of virtual ants. The colony will be placed in a playing field, where the ants will explore the field and try to find food. The ants must bring food back to the colony, so that more ants can be produced by the colony. Your colony will compete against the colonies of other students, to see which colony makes the most ants. You will need to give rules to your ants ahead of time – once the competition starts, you do not have any control over your ants – you will need to tell them everything they need to know ahead of time.

Four colonies will play at once on a field. At the beginning of each game, each of the four colonies will produce five initial ants. These ants must then find food and bring it back to the colony so the colony can produce more ants – imagine that there is a queen ant in each colony who produces baby ants when given enough food. The colony that produces the most total ants (at least 6 total) by the end of the game wins that game. If two colonies produce the same number of ants, then the colony that did so first wins that round. The winning colony will then continue to another game, competing against other winning colonies, until there is a single victorious colony and the competition is over. The more food your ants bring back to the colony, the more new ants will be produced, and the more likely you are to win the game.

A playing field is made up of squares – like spaces on a checkerboard. Ants can move from one square to another as they explore the playing field. The playing field will include anthills (the sites of different colonies), pebbles (which block your ants' movement), and food.

Ants need food to survive. Every second, 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, 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 a chemical called a pheromone, 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 – this allows the ant to move one square forward in the direction it is currently facing
- Turn – this allows the ant to change the direction that it is facing
- Pick up food that is on its same square
- Drop food that it is holding, for instance, on top of the ant's anthill to feed the colony
- Eat food that it is 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 enemy ant 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 ant
- Determine if it was just blocked from moving forward, for instance because a pebble was in the way

You gives rules to your ant by using different combinations of the above commands. Below is an example of ant rules. It has two parts:

1. The very **first line** specifies the name of your ant colony, for example "BasicAnt". Your ant colony's name must have 8 or fewer letters in its name (e.g., MyAnt, AntMan, etc.).
2. The **remaining lines** are the actual rules that will control your ants – we will call this the ant ***program***.

```
colony: BasicAnt

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 made up of simple instructions (commands) 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 of your program.

Each ant his its own ant brain and is provided with its own copy of your program. Each ant will run this program entirely on its own – but you can have your ants talk to one another using pheromones.

Each ant follows the instructions in the program one after another, following your program from top to bottom. So, in the ant program above, when a new ant is born, the ant will start by following 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.

Comments

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. But anything after the // will be ignored by the ant – so don't place your instructions after a // or they will be ignored!

Labels

The ant programs are organized by a collection of labels. For example, the sample program for BasicAnt has labels like this:

```
Start:
```

or

```
on_hill:
```

A label is just a name (a single word only) that identifies a particular line of your program. You will use labels to organize your program, and to move around the program. For example, you may have a label "HaveFood" that you can use to organize commands that you would want your ant to perform when it has picked up some food. You may have another label "OnAntHill" that

you can use to organize commands that you would want your ant to perform when it is on the square occupied by your colony on the playing field. Your program 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 performing instructions from that line onward.

Conditions

In addition to taking actions (moveForward, pickUpFood, etc), the ant can observe its environment in the playing field. There are special names for the observations that the ant can make – these are ***conditions***. Conditions will either be ***true*** or ***false***. A true condition is one that is actually observed ant. A false condition is one that is not observed by the ant. For example, the condition `i_am_standing_on_food` is true if the ant is currently standing on a square that contains food – it is false if the ant is not currently standing on a square that contains food. You can think of it as a question that the ant will ask about its environment, and you can use the answer to this question to change the ant's behavior. Your program can use what are called ***if*** statements to go to different labels based on these conditions. This is accomplished with the ***goto*** command. The goto command changes the next instruction to be performed by a particular ant. For example:

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

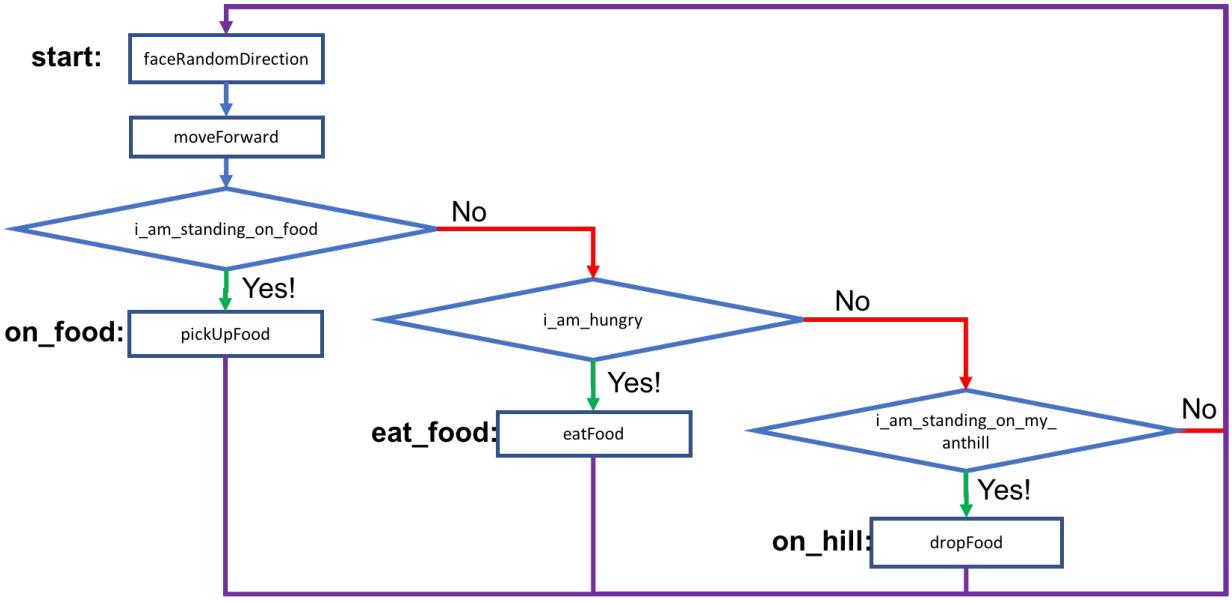
Each if statement may check a single condition, e.g., `i_am_standing_on_food`, and if that condition is met (true), then the if statement will use the goto command to tell the ant to perform the instruction that follows the specified "label:". For example, the following fragment of a program will cause 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 would go here

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
```

When 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 (faceRandomDirection) and then attempting to move forward (moveForward).

It is helpful to try and visualize what your ant program is doing. One way to do that is to create a diagram that shows what your program is doing. For example, the BasicAnt program can be draw with the following graph:



Here, the rectangles are actions (faceRandomDirection, moveForward, etc) and the diamonds are conditions. This code basically has the ant move randomly around, checking to see if it is on top of food, hungry, or on top of their anthill.

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, 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 *someNumber*

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.

Each ant can drop three different types of pheromones – specified by the *someNumber* parameter. The s*omeNumber* parameter may be the values 1, 2, or 3. So emitPheromone 1 produces a different pheromone than emitPheromone 2. Ants can use these different pheromones in creative ways to mark the ground and communicate with other ants.

rememberPheromone

This command tells the ant to sniff the pheromone scent on its current square and remember what type of pheromone it is – whether it is the 1, 2, or 3 value specified to the emitPheromone command – and how strong the pheromone is. Pheromones dissipate over time, and get weaker over time – so stronger pheromones have more recently been made by another ant. The ant can only remember one pheromone scent at a time – but this command can be useful to allow the ant to remember the strength of a pheromone and compare it to other new scents (see the if statement section for more details).

faceRandomDirection

This command causes your ant to face a random direction (e.g., up, down, left or right). Ants cannot move diagonally.

goto *someLabel*

Your ant can use the goto command to jump to a particular label in your program. This command, when executed, will immediately transfer control to the next command after that specified label. 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 – think of it as flipping a coin to make a decision. Maybe the ant is deciding whether to turn left or right at a pebble that it is in the way. It can flip a coin to decide which way to turn. The command will generate a random number between 0 and the *someNumber* parameter 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. Ants can only face one of four directions (north, south, east, and west).

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
last_pheromone_stronger
same_pheromone_type

For example, the statement:

if **i_smell_danger_in_front_of_me** then goto changeDirection

will check if the ant smells an enemy ant (from a different colony) 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 an enemy ant 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 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 currently 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 home colony's 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). 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
```

if **last_pheromone_stronger** then goto *someLabel*

Checks the last pheromone remembered by an ant (by the rememberPheromone command) against the pheromone currently in the square in front of an ant to see if the remembered pheromone is stronger – if it is, then the program will goto the specified label.

if **same_pheromone_type** then goto *someLabel*

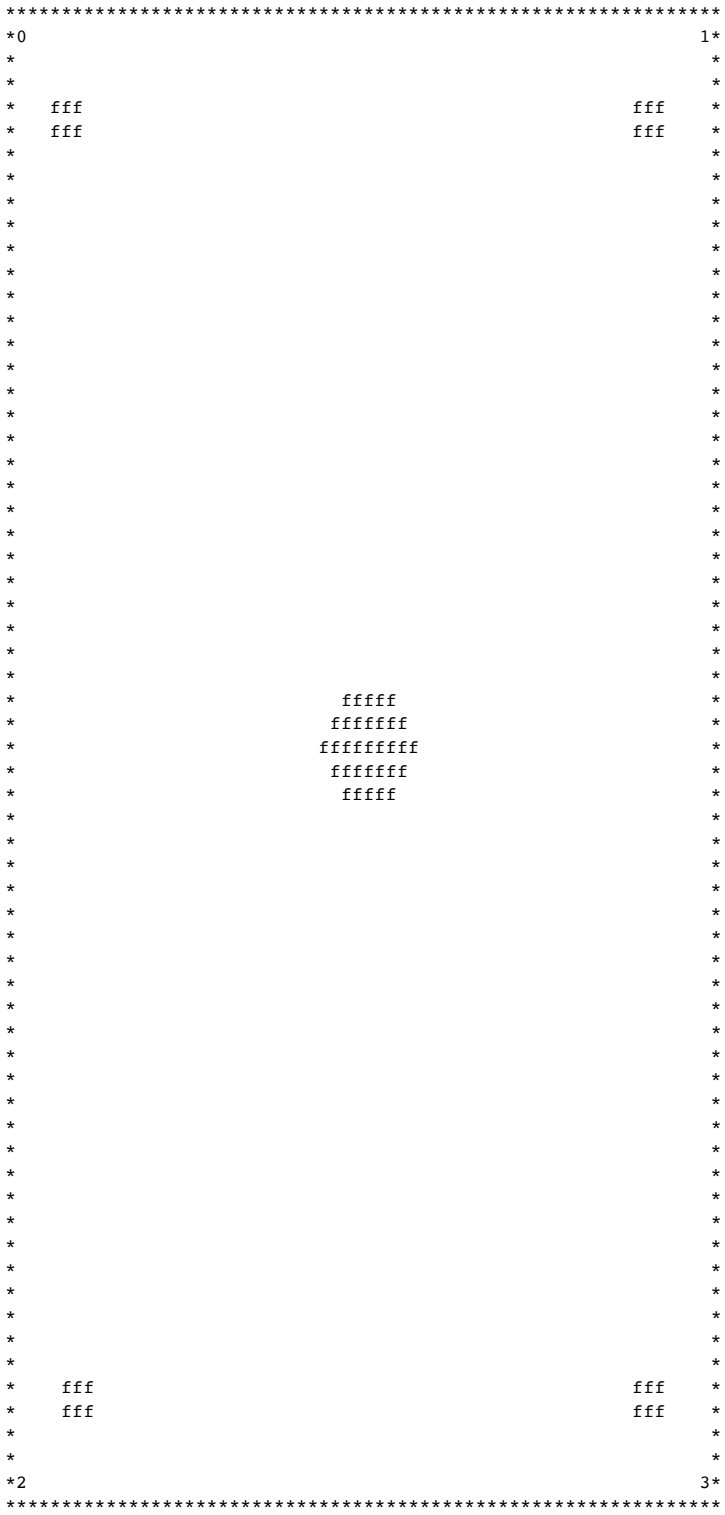
Checks the last pheromone remembered by an ant (by the rememberPheromone command) against the pheromone currently in the square in front of an ant to see if the remembered pheromone is the same type – if it is, then the program will goto the specified label.

The Field Data File

You can test your ant program in multiple different playing fields. The project web page will have a few pre-made playing fields for you to try out. But feel free to create your own playing fields to test your ant's logic.

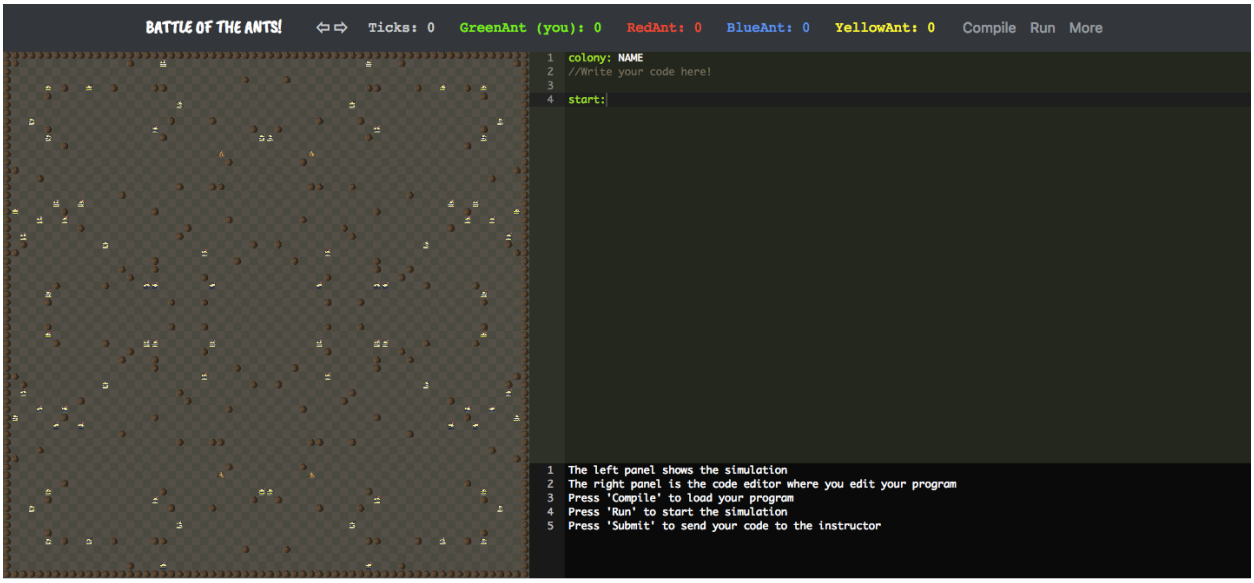
Playing fields are defined in a field data file. Each field data file is a simple text file - you can edit it with Notepad on Windows, or TextEdit on a Mac, for example.

The field data file must be exactly 64 characters wide, by 64 characters high. A character for this file must be either a space ( ), an asterisk (*), a number from 0 to 3, or an f. The '*' characters designate pebbles which block movement of the ants, '0', '1', '2' and '3' specify the location of the four colonies' anthills, and the 'f' characters specify piles of food. The top and bottom rows and the left and right columns of the field must contain a pebble at each position. 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. Here is an example field:

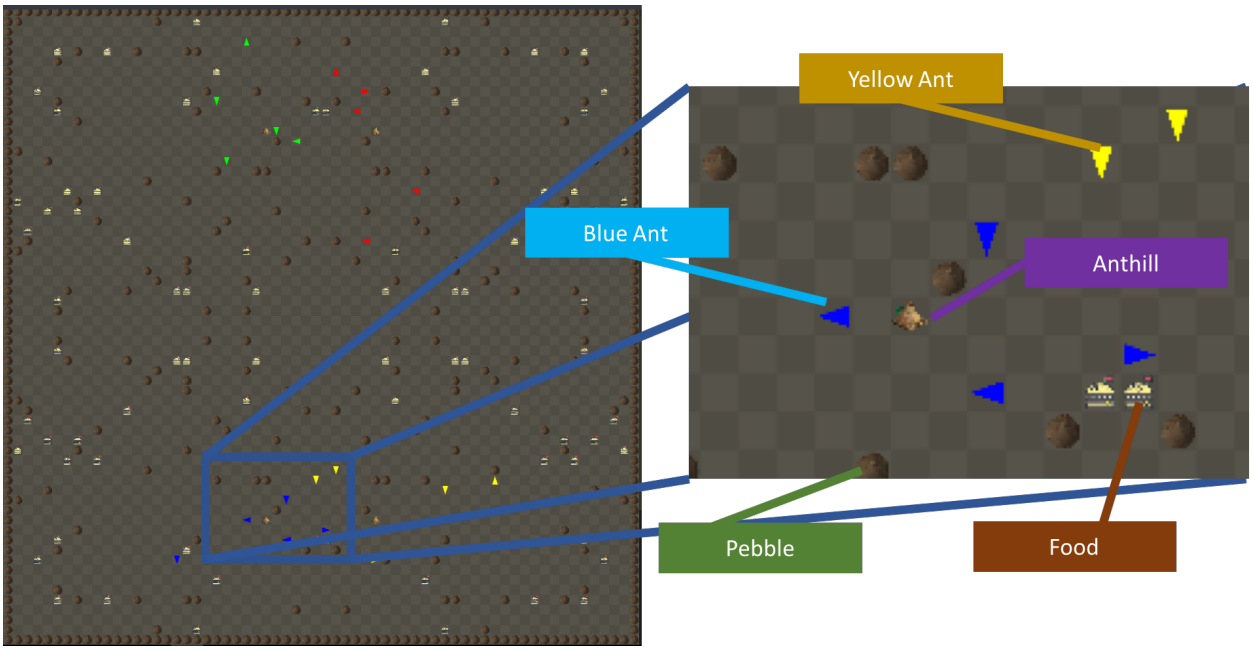

0
1
fff
fff
fff
fff
fffff
fffffff
fffffffff
fffffff
fffff
fff
fff
fff
fff
2
3

Trying out Your Ants - Running The Competition

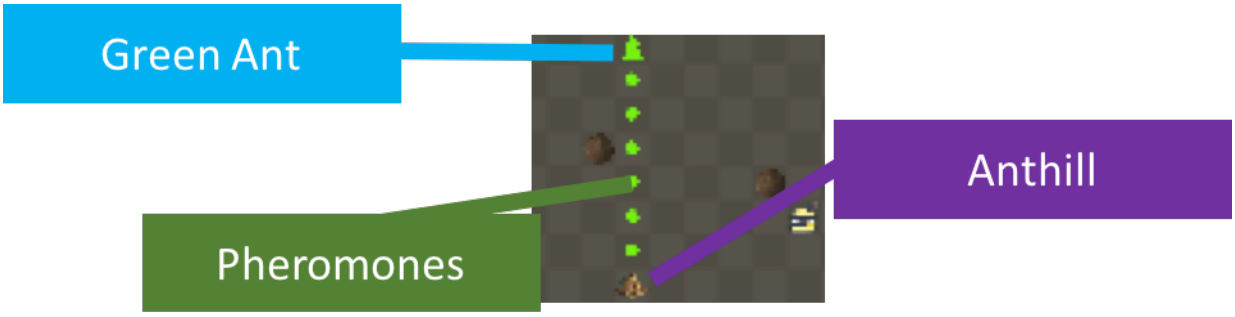
The ants programming environment is located at **http://kentercanyon.org/ants/** there is also a link from the Battle of the Ants page on the Kenter Web Site. When you go to the environment, you will see a page like this:



The left part of the window shows the simulated world of the ants – a picture of what is happening in the playing field as the ants battle for food. The following zoomed in picture shows the pebbles, anthills, ants, and food that make up the playing field.



Pheromone trails are also shown in the simulated world – but the type and strength of the pheromone is now shown. Pheromone trails appear as dots that are the color of the ant that created the trail:



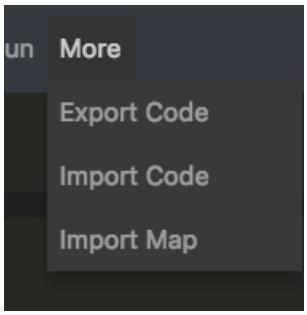
Above the simulated world, you will see the score for each of the four ant colonies – the score is the total number of ants that are currently alive in the world. The more ants you have at one time, the higher your score. For example, the scoreboard shown below shows that after 52 "Ticks" (the clock in the world of the ants), the Green Ant has a score of 8 (there are 8 green ants) and the other ants (red, blue, and yellow) still only have a score of 5 (there are only 5 of each type of ant for red, blue, and yellow).

The upper right part of the window has your code. You may write the code directly in this window, or you can **import** your code from your computer (described below). There are three buttons above the code: ***Compile***, ***Run***, and ***More***.

Compile asks the computer to check over your code and make sure it is ready to run. For example, if you used an instruction that doesn't exist (like "winTheGame"), you would see an error that says "Invalid command" in the lower right part of the window. This is shown below:

The ***Run*** button starts the simulation – the ants on the left will begin to move and the Ticks counter will start to increase. The simulation will run for 2000 Ticks total, and then stop to declare a winner. The highest score wins – in case of a tie, the first colony to reach that tied score will be declared the winner. If no colony has more than 5 ants, there will be no winner.

The ***More*** button pulls down a menu with three additional buttons: ***Export Code***, ***Import Code***, and ***Import Map***, as shown below.



Export Code saves your ant program to your local computer. ***Import Code*** brings a saved program from your local computer into the upper right window of the programming environment. You can use ***Export Code*** and ***Import Code*** to save and restore your program – the programming environment does not save the program on its own, so you need to be sure you save your efforts! ***Import Map*** allows you to use any custom playing field you want in the programming environment. See the Playing Field section of this document for more details on how to format the playing field file.

Hints

1. **Build your ant incrementally:** Don't try to build the perfect, complex ant all at once. Instead, build simple ant 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 food 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!