**Geppetto Language Tutorial**

Team 22

Paul Holmes, Mitchell Aharon

1. Content

[1. Contents 3](#_Toc355681760)

[2. Hello World 3](#_Toc355681761)

[3. Count to Ten 3](#_Toc355681762)

[4. Predator-Prey Simulation 4](#_Toc355681763)

[5. More Advanced Features 6](#_Toc355681764)

# Contents

# Introduction

Here we teach you how to use gepetto. We will start with a hello world program. This is just one simple function call.

# Hello World

Start by creating a file helloworld.gep.

*helloworld.gep*

rule true -> { print("hello world"); end; }

This translates program into C code. By default, the output of this stage is a C source file with the name of outfile.c. Alternatively, we can specify a file name with -o flage: "geppetto helloworld.gep -o helloworld.c". Then "gcc helloworld.c" and "./a.out". Outputs to screen:

Hello World. This program evaluated

That was a simple example to show you how to compile the program. Now we proceed to show you another program in order to get you used to the unique syntactical features of Geppetto.

# Count to Ten

*counter.gep*

int i = 0;

rule true -> { i = i + 1; extern.printf("%d", i); }

rule i == 10 -> end;

# Predator-Prey Simulation

Now we are going to do something more interesting and show you how to make a predator-prey simulations with foxes and rabbits.

We will create this foxrabbit.gep file:

property pos(int x {1-3}, int y{1-3});

property alive(boolean b);

property mood(string s {“hungry”, “vulnerable”});

entity wolf { pos(1,3), mood(“hungry”), alive(true) }

entity rabbit { pos(3,2), mood(“vulnerable”), alive(true) }

rule a.mood.hungry && b.mood.vulnerable && (a.pos.x == b.pos.x && a.pos.y == b.pos.y) -> { eat(a,b); end; }

To reiterate the basic structure of a Geppetto file, it has these basic components:

* definition of global variables - for example, int year = 2013
* declaration of properties - where you define the states that your simulation's actors can take.

For example: property alive(boolean b). This creates a property called alive with a single attribute b that is of boolean data type. Compiler knows that boolean is either true or false so no need to specify its range of values.

A second example: property position(int x{1-3}, int y{1-3}). This creates a property called position with two attributes x and y that are of integer data type. To give a property multiple data types separate each declaration with a comma.

A third example: property mood(string s {“vulnerable”, “hungry”}). This creates a property called alive with a single attribute b that is of boolean data type.

A note about Geppetto philosophy, Geppetto differs slightly from the C philosophy that declaration resembles use. There are four syntaxes to be aware of in relation to properties: declaration, initialization, evaluation, and assignment.

* declaration: property mood( enum e{vulnerable, hungry} )
* initialization: entity fox{ mood(“hungry”) } Entities have to be declared.
* evaluation: a.mood.e.hungry
* assignment: a.mood.s=”hungry”

Note that two entities, as in the following example program, having moods with the same name have two different moods. They are not objects that contain references to the same memory location. They are two different moods.

...

property mood( string s{“vulnerable”, “hungry”} )

...

entity fox { mood(e=hungry) }

entity rabbit { mood(e=vulnerable) }

We now proceed to create entities. An entity is an actor in the simulation - a person, or robot who does something and/or responds to events. The power of Geppetto comes from its ability to create entities, define state transitions, and run the simulation.

entity fox { position(1,3), mood(hungry), alive(true) }

This creates a fox with the position, mood, and alive properties. These properties are initialized to [1,3], hungry, and true, respectively. The properties and attributes assigned to an entity during its creation remain with it forever, but the values that the attributes take can change over the course of the program.

Similarly,

entity rabbit { position(3,2), mood(vulnerable), alive(true) }

creates a vulnerable rabbit living at position [1,3].

At this point, we have a predator (fox) and a prey (rabbit). There is no need to declare that we are uninterested in vulnerable predators or hungry prey. In order to ensure that entities only take on attribute values that are relevant to them, we define rules only for state transitions we are interested in for purposes of the simulation.

Every rule has three parts: a left hand side, an arrow, and a right hand side. Every left hand side is a boolean expression.

a.alive.b==true -> a.alive.b=false;

This line marks all living entities as deceased.

Implicitly, the program is filtering all entities to those which have alive as a property.

LHS of rules use a's and b's. This means that Geppetto finds a match among any two entities. So there is no need to specify that you are looking for a hungry fox and a vulnerable rabbit. Geppetto will find that on its own.

RHS of rules are known as behaviors and define state changes. In a behavior section, you can either place the intended state change, a series of state changes, call a function, or call a function defined in an external C file.

Congratulations! You are now familiar with all of the essential components of a Geppetto program: includes, entities, properties, attributes, initializations, state changes, rules, and behaviors.

# More Advanced Features

Let's now make our predator-prey simulation more advanced. In doing so, you will learn about more advanced Geppetto features.

We can create variables.

int minPosX = 1

int maxPosX = 100

int minPosY = 1

int maxPosY = 200

We give integer attributes values using these variable names.

property position(int x[minPosX, maxPosX], int y[minPosY, maxPosY])

Instead of giving the attributes a full list of values, we can give them a range

property position(int x[1 - 100], int y[1 - 200])

We can also use variables in the ranges.

property position(int x[minPosX - maxPosX], int y[minPosY - maxPosY])

Instead of just two possible values for mood, let's add a third

property mood(string s[hungry, happy, vulnerable])

entity fox1 { position(1-100,1-200), mood(hungry | happy), alive(true) }

Program will select an initial position at random from the given range 1 < x < 100, 1 < y < 200 and an initial mood at random from among hungry and happy.

We create another fox, also with initial position and mood chosen at random from the supplied range of possibilities

entity fox2 { position(1-100,1-200), mood(hungry | happy), alive(true) }

Creation of rabbits should look familiar by now

entity rabbit1 { position(3,2), mood(vulnerable), alive(true) }

entity rabbit2 { position(3,2), mood(vulnerable), alive(true) }

There is one more key section in a Geppetto file, the code section. This is where you define your own functions and behaviors, and your imagination can really flow!

Earlier, our rule right hand side looked like

... -> b.alive(false);

Now we create a behavior and place it on the RHS

... -> eat(a,b);

where eat is defined as:

eat(a,b) {

b.alive(false);

a.mood(e=happy);

}

Suppose we wanted to make our predator search for prey. We could define a search behavior as follows:

search(a) {

a.pos.x = a.pos.x + random(-1,1)

a.pos.y = a.pos.y + random(-1,1)

}

A great feature of Geppetto is that within a behavior we can filter all entities to those matching a given property and conveniently operate on those entities

behavior(a) {

forall Entities.mood(vulnerable) {

search(Entity)

}

}

Full source code of this program:

int minPosX = 1

int maxPosX = 3

int minPosX = 1

int maxPosX = 3

property position(int x[minPosX - maxPosX], int y[minPosY – maxPosY]);

property alive(boolean b);

property mood(string s[hungry, happy, vulnerable]);

entity wolf1 { pos(1-100,1-200), mood(hungry | happy), alive(true) };

entity wolf2 { pos(1-100,1-200), mood(hungry | happy), alive(true) };

entity rabbit { pos(3,2), mood(vulnerable), alive(true) };

a.mood.hungry && b.mood.vulnerable && (a.pos.x == b.pos.x && a.pos.y == b.pos.y) -> {eat(a,b); end;}