**Geppetto Language Tutorial**

Team 22

Paul Holmes, Mitchell Aharon

1. Content

[1. Contents 2](#_Toc355979162)

[2. Introduction 2](#_Toc355979163)

[3. Hello World 2](#_Toc355979164)

[4. Debug Mode 3](#_Toc355979165)

[5. Tree Mode and Debug Mode 4](#_Toc355979166)

[6. Adding Code Section 4](#_Toc355979167)

[7. Rules with Name 5](#_Toc355979168)

[8. Rule with Dynamic Condition 5](#_Toc355979169)

[9. Lack of End Statement 6](#_Toc355979170)

[10. Global Variables 7](#_Toc355979171)

[11. Recursive Function 7](#_Toc355979172)

[12. While Loop, Int Range, and Automatic Type Conversion 9](#_Toc355979173)

[13. Typical Day at School 10](#_Toc355979174)

[14. Predator-Prey 12](#_Toc355979175)

[15. Conclusion 14](#_Toc355979176)

# 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

In the tradition of K&R, we start by showing how to write a hello world program. The hello world print statement goes in the section designated for executed functions, which is in the Geppetto language’s “behavior” section. In order to do this, we need to set the three sections of the file, those of properties, entities, and rules. We create a property called rank and give it two values: sailor and captain (these values are completely arbitrary and we choose them just for the sake of demonstration). Now we have a property and we need an entity in order to be able to put it to use. Create an entity called Sam and initialize him to rank of captain. We assign the rank property to Sam. The rule in our code is an anonymous one, that is it does not have a name. The rule will evaluate to true and the behavior will execute. Then, the “end” statement will terminate execution of the program (we discuss what happens in the case where there is no “end” in section 9 below). Let us create the file helloworld.gep.

*helloworld.gep*

rule property rank(string s {"sailor", "captain"});

entity sam {rank(s="captain")};

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

We run this with

java –jar geppetto.jar helloworld.gep

and it outputs:

Running program...

hello world

End statement encountered. Terminating program.

Program execution complete.

We can see that our hello world string gets printed. The other 3 lines are built into the language and generated automatically, without us calling a print function.

# Debug Mode

Before proceeding to discuss more language features, we interject to mention that we can specify arguments to Geppetto to get a deeper understanding of what is taking place under the hood. With

java –jar geppetto.jar –d helloworld.gep

We direct Geppetto to debug mode. This shows us the sequence of events taking place inside Geppetto’s runtime engine. In our hello world, the output with “-d” is:

Running program...

Starting cycle #1

Processing rule: <anonymous>

Condition of rule <anonymous> is true, executing its behavior...

hello world

End statement encountered. Terminating program.

Program execution complete.

We can quickly read the output to follow along with Geppetto as it systematically processes rules and conditions.

# Tree Mode and Debug Mode

Alternatively, we can toggle the “-t” flag, which shows us the parse tree:

java –jar geppetto.jar –t helloworld.gep

this outputs the

{GeppettoProgram: contexts: [org.geppetto.ProgramContext@384e23c3]

propertyDefinitions: [{PropertyDefinition: name: rank; attributes:

({AttributeDefinition: name: s; type: STRING; constraint:

{AttributeConstraintStringSet: values: [sailor, captain]}})}]

entities: [{Entity: name: sam; properties: ({Property: name: rank;

attributes: [{Attribute: name: s; value: {Value: type: STRING;

intValue: 0; floatValue: 0.0; stringValue: captain; booleanValue:

false}}]})}]

rules: [{Rule: name: <anonymous>; condition: {ConstantExpression:

name: null; value: {Value: type: BOOLEAN; intValue: 0; floatValue:

0.0; stringValue: null; booleanValue: true}}; behavior:

{CompoundStatement: variables: []; statements: [{PrintStatement:

stringExpression: {ConstantExpression: name: null; value: {Value:

type: STRING; intValue: 0; floatValue: 0.0; stringValue: hello world;

booleanValue: false}}}, {EndStatement}]}}]

functionDefinitions: []

}

Running program...

hello world

End statement encountered. Terminating program.

Program execution complete.

Lastly, we can run the program with “-v” verbose mode. This shows the steps of the parser. We do not show a sample output for verbose mode due to special constraints.

# Adding Code Section

Now we add the fourth major section of a Geppetto program: the Code section. Unlike the Property, Entity and Rule sections, a code section is not necessary for a basic program to run. As we will see, its purpose is to better collect and organize the code from behaviors. In our program, still called helloworld.gep, we create a function and call it from within the behavior.

*helloworld.gep*

property rank(string s {"sailor", "captain"});

entity sam {rank(s="captain")};

rule (true) -> {printHello(); end;}

int printHello() {

print("hello world");

return 0;

}

This outputs the same as our program above:

Running program...

hello world

End statement encountered. Terminating program.

Program execution complete

# Rules with Name

Now, instead of an anonymous rule, we give it the name “always”:

property rank(string s {"sailor", "captain"});

entity sam {rank(s="captain")};

rule always (true) -> {printHello(); end;}

int printHello() {

print("hello world");

return 0;

}

The output is once again the same as above.

# Rule with Dynamic Condition

Previously, our rule just contained a “true” statement, which was always true. Now we proceed to something a little more interesting and test to see Sam’s current rank property. This convenient way to check conditions and trigger behaviors in response is one of Geppetto’s main features. As we will see, this provides a powerful way to compute behavioral simulations and artificially intelligent agents.

property rank(string s {"sailor", "captain"});

entity sam {rank(s="captain")};

rule samIsCaptain (sam.rank.s=="captain") -> {printHello(); end;}

int printHello() {

print("hello world");

return 0;

}

As our entity Sam is initialized to the property rank of “captain”, the condition returns true and the behavior executes printHello().

We note that complexity of a condition is unrelated to whether it is anonymous. We can thus have a condition such as the one above, except anonymous:

rule (sam.rank.s=="captain")

# Lack of End Statement

We now consider what happens in the case where there is no “end” statement. If we do not specify an “end”, the set of rules is, by default, evaluated 100 times before the program automatically exits. This code:

property rank(string s {"sailor", "captain"});

entity sam {rank(s="captain")};

rule (sam.rank.s=="captain") -> {printHello();}

int printHello() {

print("hello world");

return 0;

}

prints:

Running program...

hello world

and another 98 “hello world”s, which we do not show here, until finally:

hello world

Maximum number of cycles reached. Terminating program.

Program execution complete.

We point out here that a behavior consisting of two or more statements, must be encapsulated in brackets ( “{“ and “}” ), while a behavior consisting of only one statement does not need brackets. In the example above, after removing “end”, there is only one statement in the behavior and we include brackets even though we do not need to.

# Global Variables

Geppetto supports global variables. In this example, we simulate a student who, based on mood and location, will perform an activity (when we say he performs an activity, we mean that we will assign him a property representing his action). We will use an integer variable to denote the time of day and create a goToClass() function that represents the student going to class at 9am. We declare an int and initialize it to 9. Then we create a mood property, with possible values: inspired, interested, and hungry. Next we create a location property, with possible values: library, class, and cafeteria. Our student is then initialized as inspired and located in the library. At 9 am, we goes to class, and at 5pm (when t = 17) the school day, and hence our simulation, ends. Our code, *student.gep*:

int t = 9;

property mood(string s {"inspired", "interested", "hungry"});

property location(string s {"library", "class", "cafeteria"});

entity student {mood(s="inspired"), location(s="library")};

rule (t==9) -> {goToClass();}

rule (true) -> t = t + 1;

rule endOfDay (t == 17) -> end;

int goToClass() {

print("time = " + t + ". Going to class now");

return 0;

}

This produces an output of:

Running program...

time = 9. Going to class now

End statement encountered. Terminating program.

Program execution complete.

# Recursive Function

We will return to our student simulation in Section 13. For now, we take a little break from that in order to demonstrate some core capabilities of the language. Geppetto supports recursive functions, and local variables. In this program, we evaluate the Fibonacci sequence of numbers 1 through 5 using a recursive fib() function. Note also that we use a local variable, x, which is different than the global variable, also named x. Our property and entity are named position and fox, respectively.

int x = 0;

property position(int x {1-10});

entity fox {position(x=1)};

rule incrementX (true) -> { x = x + 1; print("x incremented to: " + x); }

rule callRecursiveFunction (true) -> print("fib("+ x + ") = " + fib(x));

rule (x==5) -> end;

int fib(int x) {

if (x == 1)

return 1;

if (x <= 0)

return 0;

return fib(x-1) + fib(x-2);

}

Program output:

Running program...

x incremented to: 1

fib(1) = 1

x incremented to: 2

fib(2) = 1

x incremented to: 3

fib(3) = 2

x incremented to: 4

fib(4) = 3

x incremented to: 5

fib(5) = 5

End statement encountered. Terminating program.

Program execution complete.

# While Loop, Int Range, and Automatic Type Conversion

Next, we demonstrate Geppetto’s support for while loops, property ranges, and automatic type conversion. We create a property position with possible integer values ranging from 1 to 10, and initialize a fox entity with a position of 1. The whileFunction() definition expects to receive an int as its parameter. However, the actual parameter is 4.2, a float value. When this happens, Geppetto automatically converts the float to an int (4) and executes the function accordingly.

property position(int x {1-10});

entity fox {position(x=1)};

rule callWhileFunction (true) -> whileFunction(4.2);

rule (true) -> end;

int whileFunction (int x) {

while (x < 10) {

print("x: "+x+" < 10");

x= x+1;

}

return 0;

}

And this program generates output:

Running program...

x: 4 < 10

x: 5 < 10

x: 6 < 10

x: 7 < 10

x: 8 < 10

x: 9 < 10

End statement encountered. Terminating program.

Program execution complete.

# Typical Day at School

We return to the theme of section 10 of this document, namely, simulating a student’s day at school. In this simulation, the school day begins at 9 and ends at 5 (1700 hours). Student has a calendar, implemented as the goToLocation() function, which places him in the appropriate place during the appropriate time. To summarize his schedule, he has class from 9-12 and again from 3-5, lunch between 12 and 1, and is otherwise in the library. This function also automatically terminates the program after 5pm. Our goal in performing this simulation is to see when he will study, learn, or eat. The first rule iterates the clock and has him checking his calendar. The second rule evaluates whether he is learning, which he will do if he is a learning-conducive location and learning-friendly mood. Third rule determines whether he is eating, also based on location and mood, and the fourth rule determines whether he is studying. We point out that the three activities, eating, learning, and studying, and not by definition mutually exclusive. Student may at any time be doing all, none, or any combination of them.

*student.gep*

int t = 8;

property mood(string s {"inspired", "interested", "hungry"});

property location(string s {"library", "class", "cafeteria"});

property activity(string s{"learning", "eating", "studying"});

entity student {mood(s="inspired"), location(s="library"), activity (s="studying")};

rule (true) -> {t=t+1; goToLocation(); print("time is: " + t);}

rule learn ((student.location.s=="library" || student.location.s=="class") &&

(student.mood.s=="interested" || student.mood.s=="inspired") )

-> {student.activity.s="learning"; print("learning now");}

rule eat (student.mood.s=="hungry" || student.location.s=="cafeteria")

-> {student.activity.s="eating"; print("eating now");}

rule study (student.location.s=="library")

-> {student.activity.s="studying"; print("studying now");}

int goToLocation() {

if ( (t >= 9 && t < 12 ) || ( t >= 15 && t < 17 )) {

student.location.s="class";

} else if ( t >= 12 && t < 13) {

student.location.s="cafeteria";

} else if ( t >= 13 && t < 15 ) {

student.location.s="library";

} else {

end;

}

return 0;

}

It turns out, student only ends up eating during lunch time. He is learning throughout the rest of the day, and, when not in class from 1-3pm, is studying as well.

Running program...

time is: 9

learning now

time is: 10

learning now

time is: 11

learning now

time is: 12

eating now

time is: 13

learning now

studying now

time is: 14

learning now

studying now

time is: 15

learning now

time is: 16

learning now

End statement encountered. Terminating program.

Program execution complete.

This simple example reflects Geppetto’s core competency as a language for behavioral modeling. Using its convenient syntax for defining states and entities, and performing rule-based behaviors, a basic program such as the one just discussed can easily be scaled to simulate more complicated situations.

# Predator-Prey

We construct a predator-prey simulation with a fox chasing a rabbit. Our properties reflect the basic types of animals we have: predator and prey. Even though we have no intention of allowing rabbit to hunt or fox to fear rabbit, we let “predator” and “prey” be two possibilities for “type”. To ensure that fox remain a predator and rabbit prey, we simply initialize them appropriately and only define behaviors that allow an animal to remain in its initialized state. Thus we do not have behaviors setting rabbit to predator or fox to prey.

By now, the “state” property should be easily understood, but we explain anyways. There are 5 possible states that an animal may be in: foraging, hunting, relaxing, fleeing, and dead.

We also create a virtual map of the forest with the pos (short for position) property. This property has both an x value and a y value, denoting that the forest is a two-dimensional space. Each of x and y may take integer values ranging anywhere between 1 and 10.

Next, we create the animals and initialize their properties, bringing us to the rules section. Basically, while hungry, fox will proceed to stalk rabbit and move closer to it. Eventually, fox may become in geographic proximity to rabbit and will hunt it. The rabbit’s state then changes to “dead”, the fox relaxes, and the simulation ends.

*predator-prey.gep:*

property type(string s {"predator", "prey"});

property state(string s {"foraging", "hunting", "relaxing", "fleeing", "dead"});

property pos(int x{1-10}, int y{1-10});

entity fox {type(s="predator"), state(s="hunting"), pos(x=7, y=5)};

entity rabbit {type(s="prey"), state(s="foraging"), pos(x=1, y=1)};

rule (true) -> printStatus();

rule sameLocation

(fox.state.s == "hunting" && fox.pos.x == rabbit.pos.x && fox.pos.y == rabbit.pos.y)

-> eat();

rule differentLocation

(fox.state.s == "hunting" && (fox.pos.x != rabbit.pos.x || fox.pos.y !=

rabbit.pos.y))

-> approach();

int printStatus() {

print("fox: state: " + fox.state.s + ", x=" + fox.pos.x + ", y=" + fox.pos.y);

print("rabbit: state: " + rabbit.state.s + ", x=" + rabbit.pos.x + ", y=" +

rabbit.pos.y);

return 0;

}

int eat() {

print("Gotcha!");

fox.state.s = "relaxing";

rabbit.state.s = "dead";

printStatus();

end;

}

int approach() {

print("Moving toward prey...");

if (fox.pos.x > rabbit.pos.x)

fox.pos.x = fox.pos.x - 1;

if (fox.pos.x < rabbit.pos.x)

fox.pos.x = fox.pox.x + 1;

if (fox.pos.y > rabbit.pos.y)

fox.pos.y = fox.pos.y - 1;

if (fox.pos.y < rabbit.pos.y)

fox.pos.y = fox.pos.y + 1;

return 0;

The output of this program returns:

Running program...

fox: state: hunting, x=7, y=5

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=6, y=4

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=5, y=3

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=4, y=2

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=3, y=1

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=2, y=1

rabbit: state: foraging, x=1, y=1

Moving toward prey...

fox: state: hunting, x=1, y=1

rabbit: state: foraging, x=1, y=1

Gotcha!

fox: state: relaxing, x=1, y=1

rabbit: state: dead, x=1, y=1

End statement encountered. Terminating program.

Program execution complete.

# Conclusion

In this quick tutorial, we have shown you the basic tools you need to get you own Geppetto programs up and running quickly. We have shown you how to structure a Geppetto program into 4 main sections: those of properties, entities, rules/behaviors, and code. You now know how to create entities, initialize them to desired states, and define behaviors that will change their state in accordance with runtime conditions. You also know how to define conditions in the rules that will trigger those behaviors. Furthermore, you have learned enough about other language features (recursion, integer ranges, while loops, etc.) to begin writing your own Geppetto programs. We encourage you to build upon these tutorials and create your own programs. If you are feeling really adventurous, design your own entity and place it in an interesting environment of properties and rules, and see how it behaves!