**Geppetto Language Project Report**

Team 22

Paul Holmes, Mitchell Aharon

Contents

[1. Introduction to the Geppetto Programming Language 4](#_Toc355998989)

[2. Language Tutorial 7](#_Toc355998990)

[2.1. Hello, World 7](#_Toc355998991)

[2.2. Debug Mode 7](#_Toc355998992)

[2.3. Tree Mode and Debug Mode 8](#_Toc355998993)

[2.4. Functions 9](#_Toc355998994)

[2.5. Rules with Names 10](#_Toc355998995)

[2.6. Rules with Dynamic Conditions 10](#_Toc355998996)

[2.7. The End Statement 11](#_Toc355998997)

[2.8. Global Variables 12](#_Toc355998998)

[2.9. Recursive Functions 13](#_Toc355998999)

[2.10. While Loops, Property Constraints, and Automatic Type Conversion 14](#_Toc355999000)

[2.11. A Typical Day at School 15](#_Toc355999001)

[2.12. Predator-Prey Simulation 18](#_Toc355999002)

[2.13. Tutorial Conclusion 21](#_Toc355999003)

[3. Language Reference 22](#_Toc355999004)

[3.1. Notational Conventions 22](#_Toc355999005)

[3.2. Lexical Conventions 23](#_Toc355999006)

[3.2.1. Whitespace 23](#_Toc355999007)

[3.2.2. Comments 23](#_Toc355999008)

[3.2.3. Identifiers 23](#_Toc355999009)

[3.2.4. Keywords 24](#_Toc355999010)

[3.2.5. Constants 24](#_Toc355999011)

[3.2.6. Operators 25](#_Toc355999012)

[3.3. Types and Variables 25](#_Toc355999013)

[3.3.1. Scope 25](#_Toc355999014)

[3.3.2. Types 26](#_Toc355999015)

[3.3.3. Initialization 26](#_Toc355999016)

[3.4. Type Conversions 27](#_Toc355999017)

[3.4.1. Integers and Floating Point Numbers 27](#_Toc355999018)

[3.4.2. Booleans and Numbers 27](#_Toc355999019)

[3.4.3. Strings 27](#_Toc355999020)

[3.4.4. Implicit Conversions in Expressions 27](#_Toc355999021)

[3.5. Entities, Properties and Attributes 29](#_Toc355999022)

[3.5.1. Scope 29](#_Toc355999023)

[3.5.2. Properties 29](#_Toc355999024)

[3.5.3. Entities 30](#_Toc355999025)

[3.5.4. References 30](#_Toc355999026)

[3.6. Rules, Conditions and Behaviors 30](#_Toc355999027)

[3.6.1. Scope 31](#_Toc355999028)

[3.6.2. Syntax 31](#_Toc355999029)

[3.6.3. Conditions 31](#_Toc355999030)

[3.6.4. Behaviors 32](#_Toc355999031)

[3.7. Expressions and Operators 32](#_Toc355999032)

[3.7.1. Primary Expressions 33](#_Toc355999033)

[3.7.2. Function Expressions ( () ) 33](#_Toc355999034)

[3.7.3. Entity Expressions (.) 33](#_Toc355999035)

[3.7.4. Unary Expressions (+, -, !) 33](#_Toc355999036)

[3.7.5. Multiplicative Expressions (\*, /,%) 34](#_Toc355999037)

[3.7.6. Additive Expressions (+, -) 34](#_Toc355999038)

[3.7.7. Relational Expressions (>, <. >=, <=) 34](#_Toc355999039)

[3.7.8. Equality Expressions (==, !=) 34](#_Toc355999040)

[3.7.9. Logical AND Operator (&&) 34](#_Toc355999041)

[3.7.10. Logical OR Operator (||) 34](#_Toc355999042)

[3.7.11. Assignment Expressions (=) 35](#_Toc355999043)

[3.8. Declarations 35](#_Toc355999044)

[3.9. Statements 35](#_Toc355999045)

[3.9.1. Expression Statement 35](#_Toc355999046)

[3.9.2. Compound Statement 35](#_Toc355999047)

[3.9.3. Selection Statements 36](#_Toc355999048)

[3.9.4. Iteration Statements 36](#_Toc355999049)

[3.9.5. End Statement 36](#_Toc355999050)

[3.9.6. Return Statement 36](#_Toc355999051)

[3.9.7. Print Statement 36](#_Toc355999052)

[3.9.8. Input Statement 36](#_Toc355999053)

[3.9.9. Length Statement 36](#_Toc355999054)

[3.9.10. The Random Statement 37](#_Toc355999055)

[3.10. Functions 37](#_Toc355999056)

[3.11. The Geppetto Standard Library 37](#_Toc355999057)

[3.11.1. Variables 37](#_Toc355999058)

[3.11.2. Functions 38](#_Toc355999059)

[3.12. Language Grammar 38](#_Toc355999060)

[4. Project Plan 44](#_Toc355999061)

[5. Language Evolution 46](#_Toc355999062)

[6. Translator Architecture 47](#_Toc355999063)

[7. Development and Runtime Environment 49](#_Toc355999064)

[8. Test Plan 49](#_Toc355999065)

[9. Conclusions/Lessons Learned 50](#_Toc355999066)

[9.1. Paul 50](#_Toc355999067)

[9.2. Lessons Learned - Mitchell 51](#_Toc355999068)

[9.3. Advice to Future Teams 53](#_Toc355999069)

[9.4. Advice to the Instructor 53](#_Toc355999070)

[10. Code Listing 54](#_Toc355999071)

# Introduction to the Geppetto Programming Language

The ability to simulate intelligent behavior has been a dream of computer scientists ever since computers were invented, and the need for this capability is ever increasing. One need look no further than the game console in the living room to see that modeling intelligent behavior is an everyday need even in something as commonplace and trivial as video game software: the bad guys need to behave intelligently or the game becomes easy – and boring – very quickly.

Recently, applications like Apple’s Siri and IBM’s Watson have put intelligent devices at the forefront of public attention. Advances in speech recognition, natural language processing and data search and retrieval have combined to at least give the impression that in some cases, computers are beginning to approach human levels of intelligence. But the reality is that despite progress in a few narrow domains, we still have a very long way to go before true artificial intelligence is a reality. In the meantime, the best we can do is to simulate and model intelligent behavior.

However, the challenge of coding behavior that is credibly intelligent is a daunting one. Even experienced programmers find the problem difficult, and in any case not every developer can be an expert in AI. Indeed, the people who have the domain knowledge about how a system should behave aren’t necessarily even programmers at all.

Currently, there are few other programming languages that address these needs – at least not mature, widely used languages. Perhaps the most prominent example is generally not regarded as a programming language at all: the Unified Modeling Language (UML), which is widely used to create visual models for use in object-oriented software design. With the proper tools these models may be “compiled” into object hierarchies in target languages like Java. However, few people would attempt to write an application using UML.

A more apt example is the Reactive Model-based Programming Language (RMPL), a language designed at MIT to control the behavior of spacecraft and robotic explorers like the Mars Rover, which can’t rely on direct human guidance. This language is model-based, meaning that a program uses a model of its environment to deduce the environment’s state from observable factors, and determine the proper behavior to exhibit in response. Of course, RMPL is highly domain-specific, being written for embedded systems on spacecraft, so it is not suitable as a general-purpose programming language.

That’s where Geppetto comes in. Geppetto is a computer language designed to bridge the gap between the programmers who know how to write code, and the domain experts who understand how a system should behave. Its main design goal is to simplify the process of developing an intelligent system, both for the programmer and for the domain expert.

For example, let’s say that a psychology researcher wants to conduct an experiment in fear responses. A human test subject controls an avatar in a virtual environment – say, a 2-D maze – and the computer controls “predators” that wander the maze searching for the subject. When the subject encounters the predator, the researcher measures the subject’s fear response. This experiment would be far more compelling if the predator behaved in a realistic manner, rather than, say, simply making a beeline for the test subject immediately upon encountering her. The researcher knows how a real predator would behave, but isn’t an expert programmer; the programmer knows how to write code, but isn’t an expert in behavioral modeling or AI. What’s needed is a way for the researcher to functionally describe the desired behavior without having to learn how to program, and for the programmer to be able to write the code to implement the desired behaviors without having to learn the intricacies of AI programming.

Geppetto addresses both of these issues. It has a ***simple declarative notation*** for describing the properties of intelligent “agents” and their environment which can be used by domain experts who are not necessarily programmers. Furthermore, the language ***encapsulates the details*** of its underlying behavior modeling algorithms, so a programmer who uses it need not be familiar with the intricacies of AI programming. Thus the ***code is simpler and more concise*** than if the programmer had to start from scratch without using Geppetto. Geppetto is ***model-based***, in that the descriptions of the intelligent agents and their environment constitute models which may interact with each other in unforeseen ways to produce potentially novel behaviors. This allows systems written in Geppetto to operate in environments in which not all the details are known in advance. Finally, Geppetto is a ***general-purpose programming language***, so its use is not limited to specific embedded systems like space probes – it compiles into C, so any system that can use the C programming language can use Geppetto.

Perhaps a simple code example would be illustrative at this point (keeping in mind that this example omits some statements for brevity). Let’s revisit the example of the psychology experiment above, and that we want to model an encounter between two intelligent agents: a predator and its prey. We first describe the agents and their initial conditions, and then we wind up the simulation and let it go:

predator.state(s=“hunting”);

predator.location(x=5, y=7);

prey.state(s=”foraging”);

prey.location(x=2, y=3);

rule (predator.location.x != prey.location.x

|| predator.location.y != prey.location.y) -> chase();

rule (predator.location.x == prey.location.x

&& predator.location.y == prey.location.y) -> { eat(); end; }

The idea is that the next states of the predator and prey are automatically determined by the system based on their given initial states until some stated condition is met, and then the next action is coded.

Using declarative statements, Geppetto allows domain experts such as psychologists and economists to create complex behavioral models without needing to learn the intricacies of AI programming. The language hides the low-level implementation of model construction from the programmer and executes simulations with its integrated engine. By abstracting away the details from the programmer, Geppetto helps to close the gap between domain experts who are interested in simulations and the technical experts who are able to code them.

# Language Tutorial

Here we teach you how to use Gepetto.

## 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*

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

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

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

We run this with the command line:

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.

## Functions

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 Names

Now, instead of an anonymous rule, we give it the name “always”. Rule names are useful in debug mode because it allows the programmer to see which rule is being evaluated:

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.

## Rules with Dynamic Conditions

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")

## The 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 Functions

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 Loops, Property Constraints, 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.

## A 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 Simulation

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.

## Tutorial 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!

# Language Reference

This reference takes its inspiration from the C language reference in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, so much of the terminology and some of the material covered is the same as in the book widely known simply as “K&R”. However, the overall structure of a Geppetto program is very different from that of a program written in a traditional procedural programming language like C, so before proceeding to the lower-level details of the language syntax, it’s worth spending a little time discussing that structure and what makes it different.

Whereas a program in a procedural language like C is basically a set of statements executed more or less in sequence from a specific starting point, a Geppetto program is inherently event-driven. A set of objects called ***entities*** is configured with initial values, and a set of ***rules*** is provided to describe how those entities interact. The rules specify ***behaviors*** to execute when specific ***conditions*** are met. And that is pretty much all there is to it!

Unlike most programming languages, Geppetto has no entry point: there is no main() or other function that identifies the first statements to be executed when the application starts. When a Geppetto application starts, it begins a unit of execution known as a ***cycle***. Each cycle, every rule is evaluated. A particular rule is ***triggered*** if its condition evaluates to true, in which case its behavior is executed. Once all the rules have been evaluated, the cycle ends and a new cycle begins. Barring error conditions, this process continues until a behavior containing the ***end*** statement is executed, or a preset maximum number of cycles have been executed.

It is important to note that Geppetto has limited input/output capabilities. It has basic commands for sending and retrieving text input to and from the console, but in general it is intended to be used for back-end processing. If more complicated I/O is required (for example, a 3-D graphical interface), it may be used in conjunction with a front-end module written in another language. This decoupling of front end from back end allows Geppetto to be used in a variety of different contexts without requiring built-in knowledge of those contexts, which makes it more portable and generic.

## Notational Conventions

When describing the syntax of the Geppetto language, this document adheres to a few general notational rules which are similar but not identical to those used in K&R:

* Syntactic categories are indicated by *italic* type.
* Literal words and characters are in **bold**.
* When there is a choice of possible values, the choices are listed on separate lines. (Some notations use a vertical bar | to indicate “or” in this situation, but that can be confusing because the | character may itself appear in the language.)
* Optional values are indicated by listing each possible syntax separately, rather than one syntax with an optional element.

Also, throughout the document are sections of text enclosed in square brackets [ ] starting with the words “**TO DO:**”. The text in these sections describes a feature that we would have liked to add to Geppetto, were not able to include due to time constraints.

## Lexical Conventions

A Geppetto program consists of one or more *translation-units*. One *translation-unit* must be written in the Geppetto language; any others must be written in the C language.

The C language *translation-units* must, like those of any other C program, adhere to standard C language syntax, which will not be described in this document. As noted previously, the purpose of the C language modules is to provide the application’s input/output routines (i.e., its front end), if more than simple text I/O is required.

The first step in the compilation of the Geppetto *translation-unit* is that it is broken into a sequence of lexical units called tokens. There are seven categories of tokens: whitespace, comments, identifiers, keywords, constants, string literals, and operators.

### Whitespace

Whitespace consists of the space, tab, newline, and formfeed characters. Whitespace is ignored except to separate otherwise adjacent identifiers, keywords and constants. Thus, a Geppetto statement may be spread over several lines if desired. In other words, there is no Geppetto language construct that depends on a newline character to mark its termination.

### Comments

The characters /\* begin a comment and the characters \*/ end a comment. Comments do not nest, and they do not occur within string literals. Like whitespace, comments are ignored by the compiler.

### Identifiers

An identifier is a label used to refer to the following Geppetto language constructs:

entities

properties

attributes

functions (both internal and external)

variables

Identifier names must adhere to the following rules:

* They must consist of a sequence of letters (i.e., the letters a through z, both uppercase and lowercase) and digits (i.e., the numbers 0 through 9).
* The first character must be a letter.
* Case is significant, so the identifier “abc” is considered different than the identifier “Abc”

[**TO DO:** Enforce length restrictions. Currently identifiers may be of any length supported by the underlying platform.]

The various uses of identifiers and their semantics are discussed in section 5, **Variables**.

### Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

**boolean**

**else**

**end**

**entity**

**false**

**float**

**for**

**foreach**

**global**

**if**

**input**

**int**

**length**

**print**

**property**

**random**

**return**

**rule**

**string**

**true**

**while**

[**TO DO:** Add support for **char** and **enum** keywords.]

### Constants

The constants recognized by Geppetto are integer and floating point numbers, the Boolean values true and false, and string literals.

[**TODO:** Add support for character and enumeration constants.]

#### Integers

Integer constants are signed 32-bit numbers. Unsigned integers, and octal and hexadecimal numbers, are not supported.

#### Floating Point Numbers

A floating point constant consists of an integer, a decimal point, and a fractional part consisting of an integer.

[**TO DO:** Add syntax for exponents to floats (the letter “e”, a plus or minus sign, and an integer).]

#### Booleans

Boolean constants are the keywords **true** and **false**.

#### String Literals

A string literal is a sequence of characters enclosed in double quotes. Strings may not contain newlines or double quote characters. Strings may be concatenated by use of the + operator.

Geppetto treats strings more in the manner of Java than of C. That is, strings are treated as constants, and the Geppetto program code cannot access the individual characters of a string directly. Thus, in order to change the string “abc” to “abd”, the original string may not be modified; a new string must be constructed instead.

[**TO DO:** Add support for accessing and comparing individual characters.]

[**TO DO:** Handle escape sequences for newlines and double-quotes.]

### Operators

Geppetto supports several operators. Generally they are symbols like + or =. They are discussed in detail in the section entitled .

## Types and Variables

A variable is an identifier that refers to a storage location in memory. In Geppetto, a variable’s defining characteristics are its scope, data type, and of course whatever value it is assigned.

[**TO DO**: Add support for *type qualifiers*, specifically **constant**, which would prevent a variable from being modified once initialized.]

### Scope

A variable’s scope determines the region of the program for which it is “visible”; that is, the parts of the program in which it may be properly referenced by other Geppetto code.

In a Geppetto program, variables may be global or local. Global variables are defined outside of any code block, at the start of a Geppetto program. They exist for the lifetime of the application and may be referenced by any application code. Local variables are declared within a function and only exist for the duration of that function.

In general any variable may be global or local. The exceptions are entities, properties and rules, which must always be global.

### Types

A variable’s type defines the meaning of the value found in the storage associated with that variable. There are basic types, which are fundamental data types; and derived types, which are composed of combinations of basic types. The basic types recognized by Geppetto are int, float, boolean and string; the derived types are property, entity, and rule.

#### Basic Types

#### int

An int variable contains a signed 32-bit number (i.e., an integer), as described in section 3.5.1, **Integers**.

#### float

A float variable contains a floating point number as described in section 3.5.2, **.**

#### boolean

A boolean variable contains a Boolean value (true or false) as described in section 3.5.3, **Booleans**.

#### string

A string variable contains a string literal (minus the quotes) as described in section 4.5.4, **String Literals**.

#### Derived types

The derived types are fundamental to the nature of Geppetto and so are worthy of their own sections of the document. They are described in detail in section 6, ; section 7, ; and section 11, **Functions**.

### Initialization

All variables must be initialized when they are declared. To initialize a basic type, simply assign it a value when it is declared using the = operator. The derived types are initialized in a special way depending on the type. See sections 6 and 7 for details.

## Type Conversions

Some statements may cause the conversion of a value from one data type to another. This section describes such conversions.

### Integers and Floating Point Numbers

When a float is converted into an int, the fractional part is discarded. When an int is converted to a float, the fractional part is zero.

### Booleans and Numbers

When an int or float is converted into a boolean, the result is **false** if the number is zero and **true** otherwise. When a boolean is converted into a number, the result is 0 is the boolean is **false** and 1 if the boolean is **true**.

### Strings

The int, float and boolean data types may be implicitly converted into strings, and vice-versa. In the former case a string representation of the value is created, and in the latter case Geppetto tries to convert the string into a value of the appropriate type. For example, if a program attempts to convert the string “3.4” into a float, the conversion would result in a float with the value 3.4. Strings converted into booleans will result in a value of **true** if and only if the string is the value “true”; otherwise the result is **false**. For example, the string value “cat” would be converted into a boolean value of **false**.

If a conversion cannot be performed, an exception is thrown. For example, if a program attempts to convert the string “cat” into a float, an exception occurs.

### Implicit Conversions in Expressions

Implicit type conversions may be performed on values in expressions depending on the operator used in the expression. For example, adding a string to an int results in the int being implicitly converted into a string, and then the strings are concatenated.

The following tables summarize the type conversions performed for various operators. The value in the cell indicates the data type of the resulting value. If “X” is indicated, it means the interpreter will throw an exception upon encountering that combination of types.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ASSIGNMENT (=)** | RValue | | | |
| LValue | int | float | string | boolean |
| int | int | int | int | X |
| float | float | float | float | X |
| string | string | string | string | string |
| boolean | X | X | boolean | boolean |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ADDITION (+)** | Operand 2 | | | |
| Operand 1 | int | float | string | boolean |
| int | int | float | string | X |
| float | float | float | string | X |
| string | string | string | string | string |
| boolean | X | X | string | X |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBTRACTION (-),**  **MULTIPLICATION (\*),**  **DIVISION (/),**  **MODULUS (%)** | Operand 2 | | | |
| Operand 1 | int | float | string | boolean |
| int | int | float | X | X |
| float | float | float | X | X |
| string | X | X | X | X |
| boolean | X | X | X | X |

## Entities, Properties and Attributes

Entities are the principal data type used by a Geppetto application. They are especially significant because they represent the items being modeled by the application. They are described separately from other variables because entities are associated with a unique behavior, and because they are declared and initialized differently than other variables.

Entities are composite data types composed of one or more ***properties***. Properties are also composite data types, and are composed of ***attributes***. Attributes may be any of the basic data types. They are referred to as attributes simply to differentiate them from variables that are not part of any property.

Properties are declared separately from entities. This allows them to be reused in multiple entity definitions. For example, if we define a property called mood, we may use it in the definitions of entities bob and alice. This is especially useful if a property is complex and has many attributes.

### Scope

Properties and entities and are global variables and may only be declared at the global level. Thus, they may not be declared within a function or code block. Attributes are not declared separately, they are part of a property definition.

### Properties

A property definition consists of the **property** keyword followed by an identifier which serves as the property’s name, followed by a comma-delimited list of attributes enclosed in parenthesis **( )**.

Each attribute in the attribute list consists of a basic data type, followed by an identifier which serves as the attribute’s name, optionally followed by a constraint enclosed in curly brackets **{ }**.

The constraint for a string attribute is a comma-delimited list of string literals. The constraint for an int attribute is **either** a comma delimited list of valid integer values, **or** a minimum and maximum int values separated by a dash **-**. The constraint for a float attribute has the same format as that of an int attribute: **either** a comma delimited list of valid float values, **or** a minimum and maximum float values separated by a dash **-**. Boolean values are already constrained to **true** or **false** so no additional constraint may be specified for boolean attributes.

If no constraints are specified for an attribute, its legal values are the range of legal values for the its data type.

Here are a few examples of property definitions:

property mood(string m {“happy”, “sad”});

property alive(boolean a);

property age(int a {0-1000});

property position(int x {1-100}, int y {1-100});

### Entities

An entity definition consists of the keyword **entity**, followed by an identifier that serves as the entity’s name, followed by a list of properties enclosed in curly brackets. The syntax for each property in the property list is the name of the attribute, followed by an initializer list in parenthesis **( )**. The initializer list is a comma-delimited name=value pairs corresponding to the attributes defined for that property. If a property has only one attribute, the parenthesis and attribute name may be omitted.

Note that each attribute of each property in an entity **must** be initialized with a value when an entity is declared.

Here are a few examples of entity declarations:

entity rabbit {position(x=5,y=10)};

entity bob {age(a=35), mood(m=”happy”)};

entity alice {mood(m=”sad”), position(x=6,y=24)};

[**TO DO:** Allow omission of attribute name if property has only one attribute.]

### References

A program will often need to refer to the values of an entity’s properties. The syntax for that is: entity.property.attribute, as in:

if (bob.mood.s == “happy”) …

If the property has only one attribute, the attribute name may be omitted, as in:

if (bob.mood == “happy”) …

## Rules, Conditions and Behaviors

Rules are at the heart of a Geppetto program. They are what make it different than an application written in a programming language like C or Java. Each rule describes a condition, and a behavior to execute if that condition evaluates to true.

What makes a rule different than a simple if-then statement in another programming language is the way rules are evaluated. The mechanism used to perform this evaluation is described more fully below, but in a nutshell, each cycle Geppetto’s internal simulator evaluates every rule that has been defined *against every entity that has been defined*. In other words, a rule is triggered whenever its condition is true for *any* entity, not just a particular entity.

Rules are not variables per se and may not be assigned a value or used in statements or as operands.

### Scope

Like entities and properties, rules are global and may only be declared at the global level. It is redundant to use the **global** keyword when declaring them.

### Syntax

A rule consists of two parts, a ***condition***, which defines the circumstances under which the rule is triggered, and a ***behavior***, which defines the statements to execute when the rule is triggered.

### Conditions

Ultimately, a condition is just a Boolean expression; see section 5.2.1.3 for a description of Boolean variables and section 9 for details on the various logical operators that may be used in boolean expressions.

Due to time constraints, for now conditions are evaluated just like any other expression.

[**TO DO:** The first and most important thing currently on the Geppetto TO DO list is to add some special handling to the way conditions are evaluated. We want to add a new data type called “variant” to the language. A variant is a special kind of variable that refers exclusively to previously declared entities. Its syntax is identical to that of an entity reference, with the exception that it is prefixed by a semicolon.

The desired behavior of a variant is that a condition should be evaluated for **every** entity that could possibly replace each variant in the condition. The evaluation would be performed only for entities that have the property and attribute referenced in that variant declaration.

For example, suppose the condition is:

:a.mood == “happy”

The desired behavior is to test the condition by successively replacing :a.mood with every entity defined with the “mood” property. So if there were five entities defined with the mood property, the rule containing the condition above would be evaluated five times, once with each of those five entities.

Furthermore, variants with different names would be always be assigned different entities. For example, consider the following condition:

:a.mood == “happy” && :a.age > 25 && :a.age > :b.age

In this condition, every declared entity with both mood and age properties would be substituted for variant a, and every entity with the age property will be substituted for variant b. But the same entity would never be used for both a and b simultaneously.]

Geppetto should also have the ability to define behaviors that should be executed \*once\* if its rule evaluates to true \*any\* condition, as opposed to behaviors that should be executed for \*each\* condition with a matching entity (which is the normal behavior as described above). More generally, the application should be able to evaluate statements of first order logic which contain with the “for each” and “there exists” qualifiers. These simple seemingly additions make FOL much more powerful and flexible than simple propositional logic, which lacks that mechanic, and are crucial for defining truly useful rules. But the difficulty is in defining a simple yet still fully deterministic language syntax for supporting this mechanic.

Unfortunately this problem is very difficult to solve, and it would wreak havoc with the rather simple and elegant mechanics of expression evaluation currently implemented in the language. We desperately wanted to get this functionality into the language, but we simply lacked the time to do it. However, it’s the first item on the project’s wish list.]

### Behaviors

Behaviors are the statements that are executed if the condition of the rule in which they are evaluates to true (i.e., if the rule is triggered).

Behaviors are simply normal Geppetto statements (see section 10).

[**TO DO:** Related to the TO DO for conditions described above, the desired behavior is that any references to variant names in a behavior would be replaced with a reference to the actual entity for which the condition evaluated to true. For example, suppose this rule is defined:

rule (:a.mood=”hungry”) -> :a.action=”eat”;

Then suppose there are two entities, alice and bob, which both satisfy the condition (i.e., they each have a property mood which currently has the value “hungry”). In that case, when the rule is evaluated, the rule will be triggered twice, once for alice and once for bob, and the behaviors alice.action=”eat” and bob.action=”eat” will be executed in sequence.]

[**TO DO:** Add special rules that are triggered under specific circumstances, such as if no other rules are triggered in that cycle.]

## Expressions and Operators

An ***expression*** is a language construct which, when evaluated, produces another value. An ***operator*** is a symbol or keyword used to relate one or more values in an expression.

The precedence of expressions is the same as the order of the subsections of this section, highest precedence first. For example, the additive operators + and – have higher precedence than the relational operators > and <. Within the same subsection, operators have the same precedence, with left or right associativity specified for each subsection.

The handling of exceptions in expressions such as overflow and divide by zero is not defined by the language.

Each section below first gives the technical description of the grammar for the kind of expression being described, then a short description of its meaning. Notice that the descriptions are cumulative: the definition of a primary expression is used in the definition of a postfix expression, which is used in the definition of a unary expression, and so on.

[**TO DO:** Geppetto currently lacks definitions for operators and expressions related to pointers, arrays, and other derived types other than entities and properties. This obviously puts serious limitations on the power of Geppetto as a programming language, but these constructs introduced unacceptable complexity. These features would eventually be added into the language given sufficient time.]

### Primary Expressions

A primary expression is one of a handful of language constructs that may serve as the basis of an expression. The other expressions described in this section are all elaborations on these fundamental expressions. In other words, ultimately every expression contains one of these constructs.

Primary expression are identifiers, constants (including string literals), or expressions in parenthesis.

### Function Expressions ( () )

Function expressions are left associative.

A function is a function expression, which is the function name, followed by a potentially empty comma-delimited list of arguments. Arguments may only be the primitive data types (int, float, boolean and string). They are passed by value. See the section entitled **Functions** for more details.

### Entity Expressions (.)

An entity expression is an identifier followed by a dot followed by an identifier, optionally followed by another dot and another identifier. In Geppetto, this kind of expression refers exclusively to entities. The first identifier refers to the entity name, the second identifier is a property name, and the third identifier, if present, is an attribute name.

### Unary Expressions (+, -, !)

Unary operators are right associative.

The operand of the unary plus and minus operators must have arithmetic type. The result of the unary plus operation is value of the operand. The result of the unary minus operation is the negative of its operand.

The operand of the unary logical negation operator must have arithmetic or boolean type, and the result is of boolean type. If the operand is arithmetic, the result is **true** if the operand is zero and **false** if it is nonzero. If the operand is boolean, the result is **true** if the operand is **false** and false if it is **true**.

### Multiplicative Expressions (\*, /,%)

Multiplicative operators are left associative.

The operands of \* and / must have arithmetic type; the operands of % must have integral type.

The \* operator denotes multiplication. The / operator yields the quotient, and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined. Otherwise, it is always true that (a/b)\*b + a%b is equal to a. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

### Additive Expressions (+, -)

Additive operators are left associative.

The result of the + operation is the sum of the two operands, and the result of the - operation is the sum of the two operands. Different types may be added; most notably, any type added to a string is implicitly converted into a string, and the resulting strings are concatenated. See the section on Type Conversions for details.

### Relational Expressions (>, <. >=, <=)

Relational operators are left associative.

The result of any relational operation is a boolean value: **true** if the relation is true or **false** if it is false. The operands may be of type int, float or string. The operands must be of the same type, except that ints and floats may also be compared. If the operands are of type string, comparison is case-sensitive.

### Equality Expressions (==, !=)

Equality expressions are left associative.

== compares whether its operands are equal, and != compares whether its operands are not equal. The result of an equality operation is a boolean value: **true** if the relation is true or **false** if it is false. The operands may be of type int, float, string or boolean. The operands must be of the same type, except that ints and floats may also be compared. If the operands are of type string, comparison is case-sensitive.

### Logical AND Operator (&&)

Logical AND operators are left associative.

The result of a logical AND expression is a boolean value: **true** if both of its operands are true, and **false** if either of its operands is false. Only boolean values may be operands.

### Logical OR Operator (||)

Logical OR operators are left associative.

The result of a logical OR expression is a boolean value: **true** if either of its operands is true, and **false** if both of its operands are false. Only boolean values may be operands.

### Assignment Expressions (=)

Assignment operators are left associative.

The left operand must be an expression capable of accepting a value (an “lvalue”), specifically a variable or an entity expression. The right operand may be any expression. Implicit type conversions may be performed if the rvalue is not the same type as the lvalue. See the section entitled **Type Conversions** for details.

[**TO DO:** Add support for \*=, /=, %=, +=, -=, !=]

## Declarations

A ***declaration*** announces the existence of a variable or function to the compiler. In Geppetto, every variable must be initialized when it is declared, so all variable declarations are also ***definitions***, meaning that they result in the allocation of storage.

Variable declarations may be global or local, depending on whether they are declared at the global scope (outside a function) or local scope (within a function). All function declarations are global.

## Statements

A ***statement*** is the smallest syntactical unit of a computer language that can stand alone (an operator, in contrast, is a smaller syntactical unit, but cannot stand alone). A program generally consists of a collection of statements.

Except as noted below, statements are executed in sequence. Statements are executed for their effect and do not have values.

### Expression Statement

An expression statement is simply an expression used as a statement. Typically this would be an assignment or function call. Note that expression statements must be terminated with a semicolon.

### Compound Statement

Compound statements are groups of statements. They exist so that several statements can be executed when only one *statement* is specified in the grammar.

The interior of compound statements is where local variables are declared. An identifier declared in a compound statement “block” exists only within that block.

### Selection Statements

Selection statements choose one of multiple flows of control. Currently the only selection statement supported by the language is the **if-then-else** statement.

This statement behaves in the usual manner: if its *expression* evaluates to **true**, the *statement* is executed. If there is an **else** clause, the *statement* following the **else** is executed if the expression evaluates to **false**.

### Iteration Statements

Iteration statements cause the flow of control to loop. Currently the only iteration statement supported by the language is the **while** loop.

The **while** loop specifies that a *statement* should continue to be executed as long as the *boolean-expression* evaluates to **true**.

[**TO DO:** Add support for the the **foreach** statement. This statement is unique to Geppetto. It executes its *statement* once for every entity that has been defined. The specified *identifier* is given the value of a different entity in each pass through the loop.]

**[TO DO:** Add support for for-loops and do-while loops.]

### End Statement

The **end** statement causes the application to exit immediately.

### Return Statement

The **return** statement sets the return value of the current function to the given expression, which must match the function’s declared type, and immediately exits the current function.

### Print Statement

The **print** statement prints the value of the given expression to the console.

### Input Statement

The **input** statement is an expression that returns a string value typed by the user from the console. Input is accepted when the user presses the enter key. The program waits indefinitely for the input statement to complete.

### Length Statement

The length statement is an expression that returns the length of the given string.

### The Random Statement

The random statement is an expression that returns a random value. There are several possible syntaxes for this statement; see the Language Grammar for details.

## Functions

This section of the document summarizes the various rules that apply to functions.

Functions are expressions in that they have a type and produce a value; but they are like statements in that when their value is requested, the result is that statements are executed.

All function arguments are passed by value: their values are copied, and changes to the values of the parameters in the function do not affect the arguments from which they were copied. Arguments are effectively local variables that have the scope of the function in which they are declared. Like other local variables, a function argument “hides” a global variable of the same name. Arguments are converted, when necessary and legal, to the types of the parameters in the declaration.

All functions have a type and must return a value of that type upon exiting via the **return** statement. Failure to do so will result in a runtime exception. If you do not wish to return a value from a function, simply declare it as type **int** and return a value of 0.

Geppetto supports recursive function calls.

[**TO DO:** Add support for external functions (i.e., functions not written in Geppetto).]

## The Geppetto Standard Library

Geppetto has a small number of built-in functions and variables designed to make the language more useful.

### Variables

#### int cycles

cycles is a predefined global int variable that counts the number of cycles that have been executed. When a Geppetto program starts, it is initialized to zero, and is incremented by one at the end of each cycle.

#### int maxCycles

maxCycles is a predefined global int variable the specifies the maximum number of cycles a Geppetto program will execute before exiting. This is to prevent applications that do not have a properly defined rule containing the **end** statement from running forever. At the end of each cycle, if the cycles variable is greater than or equal to maxCycles, the program exits. The default value of maxCycles is 100, but this can be changed at any time.

### Functions

#### string input()

The input function inputs a string from the console. It causes the program to wait until the enter key is pressed.

#### int length(string s)

The length function returns the length of the given string.

#### random

The random function randomly selects a value from among those passed as parameter. There are three variants:

**int random(int** i1**, int** i2**, …);**

**float random(float** f1**, float** f2**, …);**

**string random(string** s1**, string** s2**, …);**

[**TO DO:** Define more random functions, such as those that can choose one from a range of values.]

## Language Grammar

*program:*

*variable-declaration-list property-definition-list entity-declaration-list rule-declaration-list function-definition-list*

*variable-declaration-list:*

*variable-declaration*

*variable-declaration-list variable-declaration*

*variable-declaration:*

*type-specifier identifier* **=** *assignment-expression*

*property-definition-list:*

*property-definition*

*property-definition-list property-definition*

*property-definition:*

**property** *identifier* **(** *attribute-definition-list* **)**

*attribute-definition-list:*

*attribute*

*attribute-list* **,**  *attribute*

*attribute-definition:*

*type-specifier* *identifier*

*type-specifier identifier* **{** *attribute-constraint* **}**

*entity-declaration-list:*

*entity-declaration*

*entity-declaration-list entity-declaration*

*entity-declaration:*

**entity** *identifier* **{** *property-list* **} ;**

*rule-declaration-list:*

*rule*

*rule-declaration-list rule*

*rule:*

**rule (** *expression* **) ->** *statement*

**rule** *identifier* **(** *expression* **) ->** *statement*

*function-definition-list:*

*function-definition-list function-definition*

*<empty-list>*

*function-definition:*

*type-specifier identifier* **(** *argument-declaration-list* **)**

*argument-type-list:  
 argument-declaration*

*argument-type-list**, argument-declaration*

*argument-declaration:*

*type-specifier identifier*

*identifier:*

*[a-zA-Z][a-zA-Z0-9]\**

*type-specifier:*

**int**

**float**

**string**

**boolean**

*attribute-constraint:*

*string-list*

*integer-list*

*integer-range*

*float-list*

*float-range*

*<empty-list>*

*string-list:*

*string-constant*

*string-list* **,** *string-constant*

*integer-list:*

*integer-constant*

*integer-list* **,** *integer-constant*

*integer-range:*

*integer-constant* **-**  *integer-constant*

*float-list:*

*float-constant*

*float-list* **,** *float-contant*

*float-range:*

*float-constant* **–** *float-constant*

*property-list:*

*property*

*property-list* **,** *property*

*property:*

*identifier* ***(****attribute-initializer-list* ***)***

*attribute-initializer-list:*

*attribute-initializer*

*attribute-initializer-list*  ***,*** *attribute-initializer*

*attribute-initializer:*

*identifier* ***=*** *constant*

*constant:*

*integer-constant*

*float-constant*

*string-constant*

**true**

**false**

*integer-constant:*

*[0-9]+*

*float-constant:*

*[0-9]+***.***[0-9]+*

*string-constant:*

**“***[^”]***”**

*expression:*

*assignment-expression*

*assignment-expression:*

*logical-OR--expression*

*primary-expression = assignment-expression*

*logical-OR-expression:*

*logical-AND-expression*

*logical-OR-expression* ***||*** *logical-AND-expression*

*logical-AND-expression:*

*equality-expression*

*logical-AND-expression* ***&&*** *equality-expression*

*equality-expression:*

*relational-expression*

*equality-expression* ***==*** *relational-expression*

*equality-expression* ***!=*** *relational-expression*

*relational-expression:*

*additive –expression*

*relational -expression* ***>*** *additive –expression*

*relational -expression* ***<*** *additive –expression*

*relational -expression* ***>=*** *additive –expression*

*relational -expression <= additive –expression*

*additive-expression:*

*multiplicative –expression*

*additive-expression* ***+*** *multiplicative –expression*

*additive-expression* ***-*** *multiplicative –expression*

*multiplicative -expression:*

*unary-expression*

*multiplicative-expression* ***\**** *unary-expression*

*multiplicative-expression* ***/*** *unary-expression*

*multiplicative-expression* ***%*** *unary-expression*

*unary-expression:*

*function-expression*

***+*** *unary- expression*

***-*** *unary-expression*

***!*** *unary-expression*

*function-expression:*

*primary-expression*

*identifier* ***( )***

*identifier* ***(*** *argument-expression-list* ***)***

*primary-expression:*

*constant*

***(*** *expression* ***)***

*identifier*

*identifier***.***identifier*

*identifier***.***identifier***.***identifier*

: *identifier***.***identifier*

**:** *identifier***.***identifier***.***identifier*

*argument-expression-list:*

*expression*

*argument-expression-list* ***,*** *expression*

*statement:*

*expression-statement*

*compound-statement*

*selection-statement*

*iteration-statement*

*end-statement*

*print-statement*

*return-statement*

*expression-statement:*

*expression* ***;***

***;***

*compound-statement:*

***{*** *statement-list* ***}***

*statement-list:*

*statement*

*statement-list statement*

*selection-statement:*

**if** *(expression) statement*

**if** *(expression) statement* **else** *statement*

*Iteration-statement:*

**while** *(expression) statement*

*end-statement:*

**end ;**

*print-statement:*

**print (** *expression* **) ;**

*return-statement:*

**return** *expression*  **;**

# Project Plan

Because the Geppetto team only had two people, we didn’t specialize our roles like other teams did. Moreover, due to the fact that he’s already had many years of professional experience in software development, one of us (Paul Holmes) assumed a more prominent role on the project than the other (Mitchell Aharon). So rather than having a project manager, language guru, etc., it’s perhaps fair to say that our roles were more like that of mentor and protégé.

Seeing as how our team consisted of only two members, one might think that project management was a minor concern for us. Well, we thought so too at first, and that was a mistake. One of our “lessons learned” on this project is that even if just one person is working on a project, a well-developed project plan is essential.

Originally our intention was to work without the overhead and constraints of a highly structured project plan, thinking that this would enable us to work more dynamically. But the reality is that this sacrificed a disciplined approach for minimal gain. When one is subject to the constant demands of a university like Columbia – a constant barrage of exams, homework assignments, and long-term projects – it’s all too easy to put off something you don’t “have to” do right now until later, when you have free time – which, of course, means never. As a result we didn’t really get started on the project in earnest until the semester was almost over, and had to scramble just to finish.

Because our team was so small we didn’t need to modularize our approach very much. Especially in the early parts one part of the project when we had an unhurried attitude, we worked on just one or two tasks at a time, and took turns as our schedules allowed. For example, at the very beginning we had two tasks: writing the language reference and the tutorial, so we each wrote one. Then Paul became busy with other classes, so Mitchell selected the tools we needed and set up the project skeleton. Then our schedules revered, and Paul stepped in and implemented the language grammar while Mitchell worked on other things. And so on in this manner almost until the end.

Early on in the project, we were advised to get a minimal kernel of the language working as soon as possible, and from there it would be much easier to flesh things out. While this is undoubtedly true in general, it didn’t work well in our case. Perhaps due to Geppetto’s unusual design, it doesn’t really do anything unless nearly the entire language is implemented. For Geppetto to do anything at all, it has to be able to evaluate rules; for rules to evaluate, the interpreter needs to be able to evaluate expressions (a rule’s condition) and execute statements (a rule’s behavior), and the items on which rules operate (entities, properties and attributes) need to be present. In other words, to run even the simplest Geppetto “hello, world” program, both the front and back end need to be functional, the declarative parts of the language need to be present, and at least a representative sample of both expressions and statements need to be working. Well, that about 90% of the language right there!

Because of this, and because we could only work on one or two components of the language at a time anyway, our general approach was to “begin at the beginning, and go on till we come to the end: then stop.”[[1]](#footnote-1) That is, we started with the design of the grammar, and then implemented the declarative aspects of the grammar, and then the grammar for statements and expressions, and then we finished off the rest of the interpreter’s front end, and finally the back end. In essence, we developed the code in the same order in which data passes through the interpreter.

The timeline of our project milestones is as follows:

March 27: First pass at the language grammar submitted in the Language Reference Manual

April 10: Project skeleton created

April 18: GitHub repository created

April 27: Grammar for declarative elements finished

May 1: Grammar for statements and expressions finished; front end essentially complete

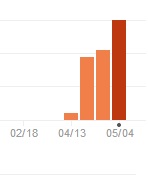
May 5: Rule evaluation working

May 7: Back end essentially complete

May 10: Code finalized

May 12: Project report submitted

Two graphs of the GitHub commit record for this project are shown below. Note the rather recent start date, and the steady increase in activity over the period shown. Other graphs available on the GitHub website indicate that, for whatever reason, Tuesdays were by far our busiest day of the week!

A final note on the subject of project management is that we did in fact use a style sheet for the code. It’s not human readable, though, unless you can digest a lengthy XML file filled with highly technical jargon. It’s a Java code formatter for Eclipse, which was only slightly modified from the default formatter that Eclipse provides, mainly to provide for longer line length before it wraps text. It’s available in the project’s root directory on GitHub if you’re really interested.

# Language Evolution

The tools used to create Geppetto were pretty much the standard tools you might expect anyone to use in the creation of a Java-based compiler:

* Eclipse was our development environment
* Ant was our build tool
* GitHub was our source code/document repository
* JFlex, a freeware Java library version of the ever-popular Lex, was our lexer generator
* BYacc/J, a freeware version of the ever-popular Yacc, was our parser generator
* JUnit was our testing platform (insofar as we used it, which frankly was not much)

The principal design of the language grammar remained remarkably consistent throughout course of the project. Admittedly that’s partially because there were only two people on the project team, so there wasn’t much in the way of conflicting input. But also, a great deal of thought went into the design of the initial language grammar before a single line of code was written.

As a result, although there were of course many minor tweaks over the course of the project, the language grammar looks more or less the same as it did in the first draft of the LRM. The main changes were the few features we had to remove from the language due to time constraints. Foremost among these was a dramatic simplification in the manner in which rule conditions are evaluated. Unfortunately that change constitutes a rather substantial detraction from the language’s utility, but the reality is that there simply wasn’t enough time to implement it properly.

On the other hand, while the language grammar didn’t change very much, the underlying development platform changed quite a bit. There were two main changes in this area, both made early on before we wrote any code: one was the switch from implementing Geppetto as a compiler to implementing it as an interpreter; the other was the switch from C to Java as the language of implementation. Both of these changes had a major positive effect on the project, the former because an interpreter is inherently much easier to implement than a full-fledged compiler, the latter because Java is (in this writer’s opinion) a much more powerful language than C due to its object-oriented capabilities and its standard class library, both of which were exploited to great effect in Geppetto’s implementation.

We also completely abandoned the idea of implementing a semantic analyzer late in the project. At first we just thought of this as an expedience, because we were running out of time and were looking for ways to cut corners. The reasoning was that as long as we used only well-formatted input files, we could do without semantic analysis, whose main purpose is to detect errors. However, as we implemented the back end, it quickly became apparent that the interpreter had to do most of the tasks of the semantic analyzer anyway.

For instance, take type checking. This is normally one of the main jobs of a semantic analyzer. But in order to implement, say, the addition operator, the Geppetto interpreter has to examine the types of its operands so it can decide how to add them. Incompatible types inevitably produce an error there, just as they would in the semantic analyzer. So in essence, the interpreter itself is doing type checking.

And that was just for starters. Eventually we implemented *all* the tasks that we had previously earmarked for the semantic analyzer in the interpreter. The main difference is that these tasks are performed at runtime rather than compile time – but for an interpreted language like Geppetto, those things happen at essentially the same time anyway. So we didn’t lose much by cutting out the semantic analyzer. Of course, this does mean that some errors aren’t detected until runtime, and if not all parts of a Geppetto program are exercised every time it runs, that means some errors can remain undetected for a long time. Nevertheless, it seemed like a worthwhile tradeoff. Given more time, we could plug in a semantic analyzer without affecting the language’s syntax or functionality at all.

This also highlights a very significant change in approach we made during the course of the project: a gradual shift from tying to do things the way we thought they were “supposed” to be done, to doing them the way we felt made the most sense. For example, initially we thought we “had to” do a semantic analyzer – after all, that’s one of the boxes in the block diagram of a compiler in the Dragon Book! But eventually we realized we could do much the same thing in a different way. In short, we learned to trust our instincts.

# Translator Architecture

Geppetto follows the typical architecture of a Lex/Yacc-based interpreter.

As part of the coding process (i.e., not at runtime), the tokens used in a Geppetto program are described in a file named lexer.flex. JFlex reads this file and produces a generated class file called Yylex.java. The Yylex class serves as the lexer for the Geppetto interpreter.

Likewise, the language grammar is described in a file called parser.y. BYacc/J reads this file and produces a class file called Parser.java. Not surprisingly, the Parser class serves as the parser for the Geppetto Interpreter.

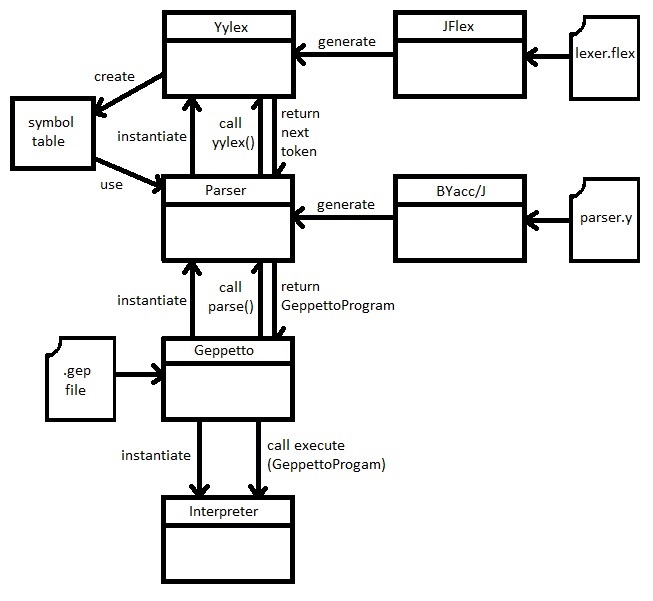
Two other key classes are Geppetto, which holds the interpreter’s main() function and is responsible for kicking off the whole process, and Interpreter, which performs the execution of the parsed Geppetto program.

The program flow is as follows:

1. The Geppetto class is invoked like any Java program. The input file, which contains the Geppetto program to be executed, is a command-line argument.
2. The Geppetto class instantiates the generated Parser class and calls its parse() function with the input file as a parameter.
3. The Parser instantiates the generated Yylex class with the input file in its constructor. The parser then proceeds to parse the input file, calling the YYlex class’s yylex() function each time it needs the next input token. The lexer uses a simple symbol table to pass strings (identifiers and string literals) to the parser, although the symbol table is later abandoned when its contents are transferred into the GeppettoProgram object model.
4. As it parses the input file, the Parser incrementally populates a GeppettoProgram object, which is essentially Geppetto’s version of an abstract syntax tree.
5. When parsing is complete, the Parser’s parse() function finally exits and returns control to Geppetto, with the constructed GeppettoProgram object as its return value.
6. The Geppetto class instantiates the Interpreter class and calls its execute() function with the GeppettoProgram object as a parameter.
7. The Interpreter executes the parsed Geppetto program stored in the GeppettoProgram object. Voila.

Most of the intelligence in the interpreter resides in the domain classes created during the parsing process (e.g., the Value, BinaryExpression, or SelectionStatement classes). Every Expression knows how to evaluate itself, and every Statement knows how to execute itself. Thus, the most complex part of the project was correctly building the abstract syntax tree, which in our case was the GeppettoProgram object. Once that was built, the rest of the interpreter’s code was almost trivial. As a result, the high-level “controller” classes (Geppetto and Interpreter) are actually very simple.

Here is a decidedly non-UML-compliant component diagram of the basic Geppetto architecture:



# Development and Runtime Environment

As noted earlier, Eclipse was our primary development platform and Ant was our build tool. However, because Eclipse automatically builds Java files, it’s only necessary to use Ant to invoke JFlex and BYacc/J to generate the lexer and parser classes, respectively, when the grammar files change. In that case Eclipse doesn’t detect that it needs to automatically recompile the newly generated Java source files, so we also added a <javac> task to the Ant build to compile the project whenever the grammar files are regenerated.

There is a way to specify that Ant should invoke Eclipse’s built-in Java compiler instead of javac. This would be preferable because then Eclipse could track compile errors and so forth like it does when it normally compiles Java files. However, this would require custom configuration in Eclipse outside the project (the classpath of Eclipse’s Ant feature would have to be amended to include Eclipse’s java compiler), and it would also make the build file highly un-portable, so we didn’t use that syntax. Most of the time when we regenerated the grammar classes, we just forced Eclipse to rebuild the project by doing a “Clean” operation immediately afterwards. It’s an extra step, but that way we could still take advantage of Eclipse’s error tracking features.

The Ant build file also contains a build target for creating geppetto.jar, though we didn’t need to use that during the normal development process because Eclipse can invoke (and debug) a Java program directly, without having to build it into a jar file first. As usual, it is extremely helpful to be able to walk through the code in Eclipse’s debugger when there is an error, because you can see the values of all the variables, the call stack, etc.

The Ant build file for the project is available for review in the project’s root directory on GitHub.

Assuming you don’t have the Geppetto project open in Eclipse, the way to a execute Geppetto program is via its jar file. There are many ways to invoke a Java program in a jar file; the surest way, if not the most user-friendly, is from the command line. To run a Geppetto program in this manner, use a command line similar to the following:

java –jar geppetto.jar myprogram.gep

This will invoke the Geppetto interpreter on myprogram.gep.

# Test Plan

This is one area where the Geppetto project team probably lags behind most other teams.

It wasn’t until the last two weeks or so or the project that we could even run a Geppetto program end-to-end. This is partially because, as noted in the chapter on Language Evolution, due to Geppetto’s design we had to have almost *everything* working in order to get *anything* working. And of course it was also because we started coding the project so late. In any case, there wasn’t really much of anything functional that could even *be* tested until the project was almost complete.

Moreover, the most complex part of the interpreter’s code, and the part that took the longest to get right, was the nitty-gritty detail of parsing the language grammar. Since parsing takes place in generated code, it is difficult to devise an automated, repeatable test for it. It is certainly possible and desirable to do so, but we simply lacked the time and resources to write such tests. If we had done so, I doubt we would have finished the project.

Therefore, the majority of the testing for the grammar and the parsing process took the form of writing an example of the grammar to be tested in a sample input file, manually running the project, and seeing whether the parser accepted the input or produced an error. As noted earlier, our development process was linear and cumulative, so as we proceeded we generally just added more and more complex syntaxes to the sample input file, and this served as a sort of quasi-regression test.

This surely would not have worked if our team had been larger. But when only one or two people are making changes at a time, it’s (relatively) easy to know what might be affected when a change is made and to devise an appropriate manual test for it.

All that said, we are (as of the time this document is being written) attempting to retrofit JUnit tests into the project, particularly for the Expression classes. These are good candidates for unit testing because they contain a large percentage of the interpreter’s intelligence, and more importantly, because they are highly modular.

And we did use one technique for testing that frequently comes in handy when writing Java code: override the toString() method of every domain object to return its state, which basically means adding the contents of each of its member variables to a StringBuffer (with an appropriate label for each one) and then returning the contents of the buffer. If, as in our case, all the program’s data is contained within one top-level object (the GeppettoProgram), the entire state of the application can be obtained by one function call (GeppettoProgram.toString()). To see this in action, you can run Geppetto with the –t (“tree”) flag.

# Conclusions/Lessons Learned

## Paul

I think it’s safe to say that for me, this project was a classic case of “be careful what you wish for”. My first team (of 5 people) was essentially built from leftover classmates who weren’t part of any other team, and as a result, we lacked cohesion. Everyone wanted to do a different project, and when we voted on it, each project came out with virtually the same number of votes. So I went to the professor to ask him what we should do about it. To my surprise and – at the time – delight, he suggested I start my own team with just two members. I’d be able to do my own project, and would essentially be running the show. Who wouldn’t want that? As a longtime professional coder, the increased workload wasn’t daunting – a projected “few thousand lines of code” is just a couple weeks of work, what’s the big deal?

Well… what I didn’t take into account was that CU has plenty of other things for an undergraduate student to be doing, so it wasn’t until there were only a few weeks left in the semester that I was finally able to devote the time I wanted to the project. Furthermore, I realized that there were probably good reasons why not everyone on my old team wanted to do my project idea – it turns out that it has some serious flaws, which I didn’t fully appreciate until the compiler was in a working state – that is, not until the last weeks of the semester, when it was far too late to make any serious changes to the language design.

Nevertheless, I have to say that once I finally started writing code, I was happy as a clam. It’s *fun* to create your own compiler, and it’s rather cool to see that you can implement features like local variables and arbitrary expressions and function calls, just like the big boys (e.g., C and Java).

So I think my most important lesson learned is, be careful not to bite off more than you can chew; and if you do, you’ve got no one else to blame, so try to make the best of it!

As for things I learned from the project itself… well, this isn’t my first software development project, and although a lot of my classmates seem to think it was a pretty big deal, it was actually rather small compared to most of the projects that I’ve worked on as a professional. So other than the technical aspects of creating a compiler (much of which I admit I still find baffling), there wasn’t really a ton that I learned in that regard.

On the other hand, I am still and probably always will be learning how to deal with people. One thing I learned is that good project management is a necessity even for a team consisting of only two people. I should have been more organized from the start and taken less for granted about how we were going to do things. For instance, one policy I instituted late in the project was mandatory status check-ins. We should have started those much sooner. But even if I had been the *only* person on the project, I think it would have benefited a lot from a more disciplined approach.

And one thing I have *yet* to learn is how to properly motivate people who don’t have the same interests as me (i.e., everyone else in the world). I’ve managed teams of developers before, and while that’s certainly not easy, when necessary you can always resort to the fact that you’re the boss and your subordinates have to do what you say. But when you’re dealing with peers, it’s a much different story, and I have definitely not mastered that skill by a long shot.

## Lessons Learned - Mitchell

As with any sophisticated project, communication and organization are essential, especially when coordinating a project between team members with different experience levels and skill sets. This is even more important with a team consisting only of two people. When one person misses a deadline, the whole project gets delayed. Challenges need to be predicted in advance in order to keep the rate of development up.

In order to facilitate the communication of code changes, it is important to write code clearly: this means sticking to agreed-upon formatting and style. For example, variable names were written in full-length instead of abbreviated. Moreover, using the versioning control system GitHub proved to be a more efficient way to manage updates than simply emailing the changes back and forth.

Iterative development is an important strategy to make sure that existing code functions before adding new code. In the case of Geppetto, this proved to be somewhat more difficult than usual due to Geppetto’s structure of inter-related components, which requires that much of the code be implemented before components can be thoroughly analyzed. Nevertheless, it remained preferable to submit well-written working code in smaller chunks than sloppy or broken code in larger ones.

As is almost always the case with projects or any sizeable complexity, it was imperative to use a variety of tools and integrate them together. JFlex and BYacc/J were used for parsing, Ant for building, Eclipse for developing, GitHub for code sharing, and, to an extent, JUnit for testing. It was also important to organize the code into a relevant package structure and separate unrelated functionalities. For instance, having input and output managed by a controller section of the project rather than interspersed among the various domain-specific classes allows for better centralization of I/O and decouples presentation from business logic.

Code organization in a large project such as this is a balancing act between generalization and specification. As a rule, code should be written in as generalized a fashion as possible in order to be able to refactor with minimal disruption to the code. Defining method signatures in terms of parent classes and using object polymorphism are important ways to enable a developer to change the underlying code without needing to rewrite all the contexts would otherwise be affected. By anticipating that changes would inevitably need to be made at a future point, we were able to write code that was better prepared to accommodate these changes, which ended up saving time in the long run.

As a rule, code repetition should be minimized and functionalities common to multiple classes should be centralized via use of inheritance. For example, the seven different Expression types in our grammar implemented a single Expression interface.

Rather than exist as completely independent objects, the nodes of the abstract syntax tree use inheritance and overridden methods to recursively evaluate grammar statements and generate string descriptors. When Geppetto is invoked with the “-t” parameter, these descriptors are retrieved and printed by the aforementioned controller.

It is also important to plan in advance how the code is going to be tested. As compilers are modular projects where each stage depends on the output of its previous stage, it is necessary to do more than just test the program as a whole. Testing should include units and components as well. Otherwise, it is too easy for bugs to go undetected. For instance, a bug in the while-loop almost went undetected when the author mistakenly thought that the repeated execution of a certain statement was due to the encompassing while loop, when in fact it was caused by the defaulting of the language’s internal rule-looping engine (good thing Paul caught this bug!).

Using print statements to debug programs is not a good idea because it couples testing with application code and requires repetitious manual checking. If required, print statements should be wrapped in a debugging method which itself should be located in a dedicated debugging class instead being invoked directly.

One especially useful feature was allowing different versions of the app to be toggled at runtime by specifying arguments to the program. In our case, in addition to the default mode, which simply computed the output of the Geppetto source code, we also allowed for the verbose, debug, and/or tree modes. These proved very useful in the verification and testing process.

## Advice to Future Teams

I’ll get the obvious one out of the way right away: “start early”! That’s easy to say but not easy to do, especially at a place like Columbia. Like any team we surely would have benefited from an earlier start. The problem is that there wasn’t any time available earlier, so I don’t know if this advice is going to be too useful.

Other than that I think any team needs a disciplined plan and a project manager that will hold team members accountable to it. In the professional world, responsibility and accountability go hand-in-hand; one is useless without the other. But it’s a lot harder to enforce accountability in the academic world, especially since your teammates are your peers. I can offer no easy answers there. A lot of times it’s not something that even needs to be addressed, but it’s one of those things where you’d rather have it and not need it, than need it and not have it.

## Advice to the Instructor

It’s definitely true that a project really gets rolling once it’s “hello, world” is working and team members can start fleshing out the missing parts. As noted in chapter 4, this approach wasn’t totally effective in Geppetto’s case due to its unusual design, but I think it would help most other compiler projects. And every team seems to say that starting earlier would be a good idea. Besides, if other CS majors are like me, they *enjoy* writing code, so giving them an early taste may get them champing at the bit to do more.

So perhaps it would be helpful to *force* teams to get started earlier by requiring some code deliverable early on, rather than not requiring any code at all until the final delivery date. For example, perhaps a week after the LRM document’s due date, maybe each team should be required to show their front end parsing a few words of the language grammar. I don’t know… *something* to coerce us into having some working code earlier in the semester rather than waiting until the last moment, as we students are wont to do.

# Code Listing

Rather than cut and paste thousands of lines of code from dozens of files into this document, we were granted permission to provide the URL for the GitHub repository where the live source code for the project may be found:

<https://github.com/pwholmes/geppetto>

1. Paraphrasing *Alice’s Adventures in Wonderland*. [↑](#footnote-ref-1)