**Geppetto Language Project Report**

Team 22

Paul Holmes, Mitchell Aharon

Contents

[2. Introduction 3](#_Toc355740846)

[3. Language Tutorial 3](#_Toc355740847)

[4. Language Reference 3](#_Toc355740848)

[4.1. Notational Conventions 4](#_Toc355740849)

[4.2. Lexical Conventions 4](#_Toc355740850)

[4.2.1. Whitespace 4](#_Toc355740851)

[4.2.2. Comments 5](#_Toc355740852)

[4.2.3. Identifiers 5](#_Toc355740853)

[4.2.4. Keywords 5](#_Toc355740854)

[4.2.5. Constants 6](#_Toc355740855)

[4.2.6. Operators 6](#_Toc355740856)

[4.3. Types and Variables 6](#_Toc355740857)

[4.3.1. Scope 7](#_Toc355740858)

[4.3.2. Types 7](#_Toc355740859)

[4.3.3. Initialization 7](#_Toc355740860)

[4.4. Type Conversions 8](#_Toc355740861)

[4.4.1. Integers and Floating Point Numbers 8](#_Toc355740862)

[4.4.2. Booleans and Numbers 8](#_Toc355740863)

[4.4.3. Strings 8](#_Toc355740864)

[4.4.4. Implicit Conversions in Expressions 8](#_Toc355740865)

[4.5. Entities, Properties and Attributes 9](#_Toc355740866)

[4.5.1. Scope 10](#_Toc355740867)

[4.5.2. Properties 10](#_Toc355740868)

[4.5.3. Entities 10](#_Toc355740869)

[4.5.4. References 11](#_Toc355740870)

[4.6. Rules, Conditions and Behaviors 11](#_Toc355740871)

[4.6.1. Scope 11](#_Toc355740872)

[4.6.2. Syntax 11](#_Toc355740873)

[4.6.3. Conditions 11](#_Toc355740874)

[4.6.4. Behaviors 12](#_Toc355740875)

[4.7. Expressions and Operators 13](#_Toc355740876)

[4.7.1. Primary Expressions 13](#_Toc355740877)

[4.7.2. Function Expressions ( () ) 14](#_Toc355740878)

[4.7.3. Entity Expressions (.) 14](#_Toc355740879)

[4.7.4. Unary Expressions (+, -, !) 14](#_Toc355740880)

[4.7.5. Multiplicative Expressions (\*, /,%) 14](#_Toc355740881)

[4.7.6. Additive Expressions (+, -) 14](#_Toc355740882)

[4.7.7. Relational Expressions (>, <. >=, <=) 15](#_Toc355740883)

[4.7.8. Equality Expressions (==, !=) 15](#_Toc355740884)

[4.7.9. Logical AND Operator (&&) 15](#_Toc355740885)

[4.7.10. Logical OR Operator (||) 15](#_Toc355740886)

[4.7.11. Assignment Expressions (=) 15](#_Toc355740887)

[4.8. Declarations 15](#_Toc355740888)

[4.9. Statements 16](#_Toc355740889)

[4.9.1. Expression Statement 16](#_Toc355740890)

[4.9.2. Compound Statement 16](#_Toc355740891)

[4.9.3. Selection Statements 16](#_Toc355740892)

[4.9.4. Iteration Statements 16](#_Toc355740893)

[4.9.5. End Statement 17](#_Toc355740894)

[4.9.6. Return Statement 17](#_Toc355740895)

[4.9.7. Print Statement 17](#_Toc355740896)

[4.9.8. Input Statement 17](#_Toc355740897)

[4.9.9. Length Statement 17](#_Toc355740898)

[4.9.10. The Random Statement 17](#_Toc355740899)

[4.10. Functions 17](#_Toc355740900)

[4.11. The Geppetto Standard Library 18](#_Toc355740901)

[4.11.1. Variables 18](#_Toc355740902)

[4.11.2. Functions 18](#_Toc355740903)

[4.12. Language Grammar 19](#_Toc355740904)

[5. Language Evolution 25](#_Toc355740905)

[6. Translator Architecture 25](#_Toc355740906)

[7. Development and Runtime Environment 25](#_Toc355740907)

[8. Test Plan 25](#_Toc355740908)

[9. Conclusions 25](#_Toc355740909)

[10. Code Listing 25](#_Toc355740910)

# Introduction

# Language Tutorial

# Language Reference

This manual describes Geppetto, a programming language designed from the ground up to be used for behavioral modeling. This document 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.

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:*

*global-variable-declaration-list property-definition-list entity-declaration-list rule-declaration-list statement-list*

*global-variable-declaration-list:*

*variable-declaration*

*| global-variable-declaration-list variable-declaration*

*property-definition-list:*

*property-definition*

*| property-definition-list property-definition*

*entity-declaration-list:*

*entity-declaration*

*| entity-declaration-list entity-declaration*

*rule-declaration-list:*

*rule-declaration*

*| rule-declaration-list rule-declaration*

*statement-list:*

*statement*

*statement-list statement*

*variable-declaration:*

*type-specifier identifier* ***=*** *literal-value*

*type-specifier:*

**int**

**float**

**string**

**boolean**

*identifier:*

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

*literal-value:*

*int-constant*

*float-constant*

*string-literal*

**true**

**false**

*property-definition:*

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

*attribute-list:*

*attribute*

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

*attribute:*

*type-specifier* *identifier*

*type-specifier identifier* **{** *attribute-legal-values* **}**

*attribute-legal-values:*

*string-list*

*int-list*

*int-range*

*float-list*

*float-range*

*string-list:*

*string-literal*

*string-list* **,** *string-literal*

*int-list:*

*int-constant*

*int-list* **,** *int-constant*

*int-range:*

*int-constant* **-**  *int-constant*

*float-list:*

*float-constant*

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

*float-range:*

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

*entity-declaration:*

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

*property-list:*

*property-entry*

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

*property-entry:*

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

*attribute-initializer-list:*

*attribute-initializer*

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

*attribute-initializer:*

*identifier* ***=*** *literal-value*

*rule-declaration:*

**rule** *condition* **->** *behavior*

*condition:*

*boolean-expression*

*behavior:*

*statement*

*statement:*

*expression-statement*

*compound-statement*

*selection-statement*

*iteration-statement*

*end-statement*

*expression-statement:*

*expression* ***;***

***;***

*expression:*

*assignment-expression*

*assignment-expression:*

*boolean-expression*

*unary-expression = assignment-expression*

*boolean-expression:*

*logical-OR-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:*

*structure-expression*

*function-expression* ***( )***

*function-expression* ***(*** *argument-expression-list* ***)***

*structure-expression:*

*primary-expression*

*identifier.identifier*

*identifier.identifier.identifier*

*primary-expression:*

*identifier*

*literal-value*

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

*argument-expression-list:*

*expression*

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

*declaration-statement:*

*variable-declaration*

*function-declaration*

*global-variable-declaration:*

*type-specifier identifier = literal-value*

*variable-declaration:*

*type-specifier identifier = assignment-expression*

*function-declaration:*

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

*argument-type-list:  
 argument-declaration*

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

*argument-declaration:*

*type-specifier identifier*

*compound-statement:*

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

*statement-list:*

*statement*

*statement-list statement*

*selection-statement:*

**if** *(boolean-expression) statement*

**if** *(boolean-expression) statement* **else** *statement*

*Iteration-statement:*

**while** *(boolean-expression) statement*

**foreach** *identifier statement*

*end-statement:*

**end;**

# Project Plan

Because we only had two people we didn’t specialize our roles like other teams did. Moreover, due to the many years of experience he already has in software development, Paul Holmes assumed a more prominent role on the project than 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: Paul took a leadership role on most aspects of the project, and Mitchell followed his lead.

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 you have the constant demands of other classes at a university like Columbia – a constant barrage of exams, homework assignments, and other long-term projects – it is all too easy to put off something you don’t “have to” do right now until 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 it 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 sample representative 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 came 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, 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; or if you prefer, the order of the boxes in the architectural flow chart.

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

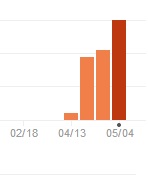
May 1: Grammar for statements and expressions finished; front end essentially complete

May 5: Rule evaluation working

May 7: Back end essentially complete

May 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 is our development environment
* Ant is our build tool
* GitHub is our source code repository
* JFlex, a freeware Java library version of the ever-popular Lex, is our lexer generator
* BYacc/J, a freeware version of the ever-popular Yacc, is our parser generator
* JUnit was our testing platform (insofar as we used it, which 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 was not 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 this just seemed like 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 in an expression, the Geppetto interpreter has to examine the types of the 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 were able to implement *all* the tasks that we had previously earmarked for the semantic analyzer in the interpreter. The only 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 really lose anything by cutting out the semantic analyzer.

This 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 diagram of a compiler in the Dragon Book! But eventually we realized we could do 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, the tokens used in a Geppetto program are described in a file named lexer.flex, which is fed as input to JFlex, which in turn produces a generated class called Yylex. This class serves as the lexer for the Geppetto interpreter.

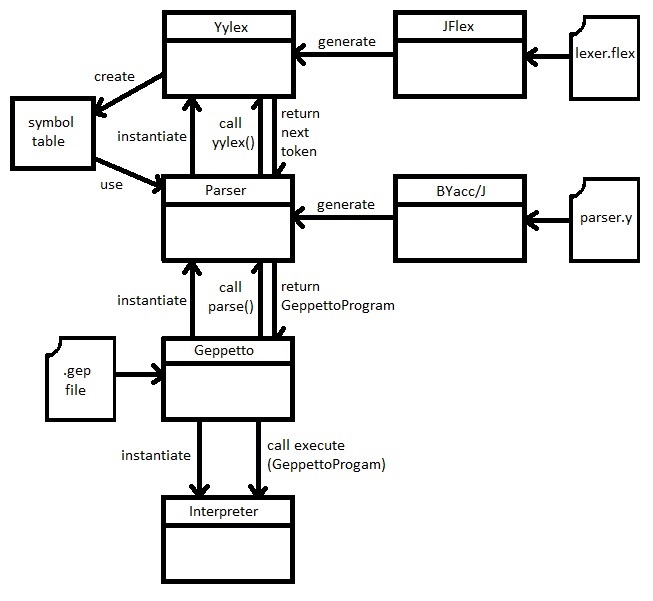
Likewise, the language grammar is described in a file called parser.y, which is fed as input to BYacc/J, which in turn produces a class called Parser. Not surprisingly, this class serves as the parser for the Geppetto Interpreter.

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

The program flow is as follows:

1. The Geppetto interpreter 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. The Parser incrementally populates a GeppettoProgram object, which is essentially our abstract syntax tree.
5. The Parser returns the constructed GeppettoProgram object to the Geppetto class, which then instantiates the Interpreter class and calls its execute() function with the GeppettoProgram object as a parameter.
6. The Interpreter executes the program stored in the GeppettoProgram object. Voila.

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 the Ant build kicks off a <javac> task 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 better in some respects because then Eclipse could track compile errors and so forth like it does when it compiles Java files normally, but it requires custom configuration in Eclipse and makes 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 entire 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 never used that during the normal development process because Eclipse can invoke (and debug) the project directly, without having to build it into a jar file first.

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 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

# Conclusions

# Code Listing

Rather than cut and paste thousands of lines of code 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:

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