**Geppetto Language Reference Manual**

Team 22

Paul Holmes, Mitchell Aharon

Contents

[2. Introduction 4](#_Toc352118806)

[3. Notational Conventions 4](#_Toc352118807)

[4. Lexical Conventions 5](#_Toc352118808)

[4.1. Whitespace 5](#_Toc352118809)

[4.2. Comments 5](#_Toc352118810)

[4.3. Identifiers 5](#_Toc352118811)

[4.4. Keywords 6](#_Toc352118812)

[4.5. Constants 6](#_Toc352118813)

[4.5.1. Integers 6](#_Toc352118814)

[4.5.2. Floating Point Numbers 6](#_Toc352118815)

[4.5.3. Booleans 6](#_Toc352118816)

[4.5.4. String Literals 6](#_Toc352118817)

[4.6. Operators 7](#_Toc352118818)

[5. Types and Variables 7](#_Toc352118819)

[5.1. Storage Class and Scope 7](#_Toc352118820)

[5.2. Types 8](#_Toc352118821)

[5.2.1. Basic Types 8](#_Toc352118822)

[5.2.2. Derived types 8](#_Toc352118823)

[5.3. Initialization 8](#_Toc352118824)

[6. Type Conversions 9](#_Toc352118825)

[6.1. Integers and Floating Point Numbers 9](#_Toc352118826)

[6.2. Booleans and Numbers 9](#_Toc352118827)

[6.3. Strings 9](#_Toc352118828)

[7. Entities, Properties and Attributes 9](#_Toc352118829)

[7.1. Scope 9](#_Toc352118830)

[7.2. Syntax 10](#_Toc352118831)

[7.2.1. Properties 10](#_Toc352118832)

[7.2.2. Entities 11](#_Toc352118833)

[7.3. References 12](#_Toc352118834)

[8. Rules, Conditions and Behaviors 12](#_Toc352118835)

[8.1. Scope 13](#_Toc352118836)

[8.2. Syntax 13](#_Toc352118837)

[8.2.1. Conditions 13](#_Toc352118838)

[8.2.2. Behaviors 14](#_Toc352118839)

[9. Expressions and Operators 14](#_Toc352118840)

[9.1. Primary Expressions 15](#_Toc352118841)

[9.2. Function Expressions ( () ) 15](#_Toc352118842)

[9.3. Structure Expressions (.) 15](#_Toc352118843)

[9.4. Unary Expressions (+, -, !) 16](#_Toc352118844)

[9.5. Multiplicative Expressions (\*, /,%) 16](#_Toc352118845)

[9.6. Additive Expressions (+, -) 17](#_Toc352118846)

[9.7. Relational Expressions (>, <. >=, <=) 17](#_Toc352118847)

[9.8. Equality Expressions (==, !=) 17](#_Toc352118848)

[9.9. Logical AND Operator (&&) 17](#_Toc352118849)

[9.10. Logical OR Operator (||) 18](#_Toc352118850)

[9.11. Assignment Expressions (=) 18](#_Toc352118851)

[10. Declarations 18](#_Toc352118852)

[11. Statements 19](#_Toc352118853)

[11.5. End statement 20](#_Toc352118854)

[12. Functions 20](#_Toc352118855)

[13. The Geppetto Standard Library 21](#_Toc352118856)

[13.1. Variables 21](#_Toc352118857)

[13.1.1. int cycles 21](#_Toc352118858)

[13.1.2. int maxCycles 21](#_Toc352118859)

[13.2. Functions 21](#_Toc352118860)

[13.2.1. int print(string s) 21](#_Toc352118861)

[13.2.2. string input() 21](#_Toc352118862)

[13.2.3. int length(string s) 22](#_Toc352118863)

[13.2.4. random 22](#_Toc352118864)

[14. Language Grammar 22](#_Toc352118865)

# Introduction

This manual describes Geppetto, a programming language designed from the ground up to be used for behavioral modeling. This document follows 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, a Geppetto program has a very different structure than a C program, so the content and structure of this document diverges significantly from that of K&R in many places. In particular, 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 C. 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 documents 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 like to add to Geppetto, but may not be 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”
* They must be 31 characters or less in length.

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**

**global**

**input**

**int**

**print**

**property**

**string**

**true**

**while**

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

## Constants

Currently the only 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, the letter **e** or **E**, and an optionally signed integer exponent.

### 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 modifying and comparing individual characters.]

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

[**TO DO:** Define the character set used by Geppetto.]

## Operators

Geppetto supports several operators. Generally they are symbols like + or =. They are discussed in detail in section 8, .

# Types and Variables

A variable is an identifier that refers to a storage location in memory. A variable has three defining characteristics: storage class, scope, and data type.

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

## Storage Class and Scope

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

In a Geppetto program, there are two storage classes: global and local. Global variables are defined outside of a code block. They exist for the lifetime of the application and may be referenced by any application code. Local variables are declared within a code block and only exist for the duration of that code block. Variables within a code block may still be defined as global by use of the **global** keyword.

Scope is essentially synonymous with storage class in that global variables are in scope throughout the entire application and local variables are in scope only in the code block in which they are declared. In addition, a variable is in scope only after the point in the code where it is declared. In other words, a program cannot reference a variable until after it has been declared.

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, rule and function.

### Basic Types

#### int

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

[**IN PROGRESS**: Determine if/how to enforce that all ints are 32 bits. If not, what is the alternative?]

#### float

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

#### boolean

A boolean variable contains a Boolean value as described in section 3.5.3, **Booleans**.

#### string

A string variable contains a string value (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, depending on their operands, cause the implicit conversion of a value from one data type to another. This section describes such conversions. If a conversion is not listed, it is not a legal conversion.

## Integers and Floating Point Numbers

When a floating point number is converted into an integer, 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, but not the other way around. When this occurs, a string representation of the value is created.

# 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 (they are referenced implicitly by the application’s rules, as described in section 7), 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. Therefore it is redundant to use the **global** qualifier when declaring them. A property must be declared before it can be used in an entity definition. (Attributes are not declared separately, they are part of a property declaration.)

## Syntax

### Properties

*property:*

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

*attribute-list:*

*attribute*

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

*attribute:*

*type-specifier* *identifier*

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

*type-specifier:*

**int**

**float**

**string**

**boolean**

*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*

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

*entity:*

**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* ***=*** *initial-value*

*initial-value:*

*int-constant*

*float-constant*

*string-literal*

**true**

**false**

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

*rule:*

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

*condition:*

*boolean-expression*

*behavior:*

*statement*

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.

What makes these Boolean expressions different is in the way they are evaluated. Any time Geppetto encounters an expression of the form a.b or a.b.c – that is, the usual syntax for referring to an entity’s attributes – it evaluates the condition for **every** entity that could possibly satisfy that expression. In other words, the entity name in the expression acts as a placeholder for **all** entities.

The evaluation is performed only for entities that have the property and attribute in the expression. For example, suppose the condition is:

a.mood == “happy”

In this case, only entities with the property “mood” are evaluated in the expression.

If there is more than one entity reference in the expression, then each reference will be replaced with every entity that “fits” the expression, except that multiple entity references using the same identifier will be replaced with the same entity. Furthermore, entity references using different identifiers will never be evaluated using the same entity for both references. 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 will be substituted for entity reference a, and every entity with the age property will be substituted for entity reference b. Furthermore, the same entity will always be used in place of every occurrence of identifier a, but the condition will never be evaluated using the same entity for both a and b.

[**TO DO:** Geppetto should 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” statements. 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. Heck, this problem is difficult to describe in English, let alone a computer language grammar! So for starters we’re not going to try to solve this problem, even though it’s fairly essential if Geppetto is going to be useful in solving any real problems. This is the first item on the TO DO 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), with the exception that any references to entity names also present in condition part of the rule are replaced with references 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 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 group of language constructs which, when evaluated, produce 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 structures other than entities and properties. This obviously puts serious limitations on the power of Geppetto as a programming language, but it was felt that these constructs would introduce unacceptable complexity at this point. These features would be among the first to be added into the language given sufficient time.]

## Primary Expressions

*primary-expression:*

*identifier*

*constant*

*( expression )*

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-expression:*

*primary-expression*

*function-expression ()*

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

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.

## Structure Expressions (.)

*structure-expression:*

*identifier.identifier*

*identifier.identifier.identifier*

Structure expressions are left associative.

A structure 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-expression:*

*function-expression*

*+ expression*

*- expression*

*! expression*

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 -expression:*

*unary-expression*

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

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

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

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-expression:*

*multiplicative –expression*

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

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

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.

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

*relational-expression:*

*additive –expression*

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

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

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

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

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, the result of the comparison is determined by the results of the C function strcmp(), which performs a case-sensitive comparison of its arguments.

## Equality Expressions (==, !=)

*equality-expression:*

*relational-expression*

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

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

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.

## Logical AND Operator (&&)

*logical-AND-expression:*

*equality-expression*

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

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-expression:*

*logical-AND-expression*

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

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-expression:*

*logical-OR-expression*

*unary-expression = assignment-expression*

Assignment operators are left associative.

The left operand must be a variable, a fully qualified reference to an attribute (i.e., entityname.propertyname.attributename), or a fully qualified reference to a property that has only one attribute. Both operands must have the same type.

[**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.

*declaration:*

*variable-declaration*

*function-declaration*

*variable-declaration:*

*type-specifier identifier = assignment-expression*

**global** *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*

# 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.

*statement:*

*expression-statement*

*compound-statement*

*selection-statement*

*iteration-statement*

*end-statement*

* 1. **Expression Statement**

*expression-statement:*

*expression* ***;***

***;***

Most expression statements are assignments or function calls. Note that expression statements must be terminated with a semicolon.

* 1. **Compound Statement**

*compound-statement:*

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

*statement-list:*

*statement*

*statement-list 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.

* 1. **Selection Statements**

*selection-statement:*

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

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

Selection statements choose one of multiple flows of control.

The **if** statements behave in the usual manner: if the boolean-expression evaluates to **true**, the *statement* is executed. If there is an **else** clause, the *statement* following the **else** is executed is the boolean-expression is **false**.

* 1. **Iteration Statements**

*Iteration-statement:*

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

**foreach** *identifier statement*

Iteration statements cause the flow of control to loop.

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

The **foreach** loop is unique to Geppetto. It executes the *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 for loops and do-while loops.]

## End statement

*end-statement:*

**end;**

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

# Functions

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

There are two kinds of functions: internal functions, which are written in Geppetto and reside in the Geppetto source file; and external functions, which are written in C and reside in other source files.

External functions may only use (as arguments and return values) data types similar to those used in Geppetto. The table below summarizes the conversions:

|  |  |
| --- | --- |
| Geppetto | C |
| int | signed long |
| float | float |
| string | char \* |
| boolean | int |

All functions, whether internal or external, must be declared prior to their use.

All 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 converted, when necessary and legal, to the types of the parameters in the declaration.

# 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

### int print(string s)

The print function outputs the given string to the console.

### 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

**[TO DO:** Please note that, as the last chapter in this unexpectedly long document, the language grammar was the last thing written right before the submission deadline, we know so it’s buggy. Obviously, the grammar is something that requires a lot of attention. We’re working on it...]

*property:*

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

*attribute-list:*

*attribute*

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

*attribute:*

*type-specifier* *identifier*

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

*type-specifier:*

**int**

**float**

**string**

**boolean**

*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:*

**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* ***=*** *initial-value*

*initial-value:*

*int-constant*

*float-constant*

*string-literal*

**true**

**false**

*rule:*

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

*condition:*

*boolean-expression*

*behavior:*

*statement*

*primary-expression:*

*identifier*

*constant*

*( expression )*

*function-expression:*

*primary-expression*

*function-expression ()*

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

*structure-expression:*

*identifier.identifier*

*identifier.identifier.identifier*

*unary-expression:*

*function-expression*

*+ expression*

*- expression*

*! expression*

*multiplicative -expression:*

*unary-expression*

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

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

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

*additive-expression:*

*multiplicative –expression*

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

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

*relational-expression:*

*additive –expression*

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

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

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

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

*equality-expression:*

*relational-expression*

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

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

*logical-AND-expression:*

*equality-expression*

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

*logical-OR-expression:*

*logical-AND-expression*

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

*assignment-expression:*

*logical-OR-expression*

*unary-expression = assignment-expression*

*declaration:*

*variable-declaration*

*function-declaration*

*variable-declaration:*

*type-specifier identifier = assignment-expression*

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

*function-declaration:*

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

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

*argument-type-list:  
 argument-declaration*

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

*argument-declaration:*

*type-specifier identifier*

*statement:*

*expression-statement*

*compound-statement*

*selection-statement*

*iteration-statement*

*end-statement*

*declaration*

*expression-statement:*

*expression* ***;***

***;***

*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;**