Gantry
Language Reference Manual

Audrey Copeland (asc2182), Walter Meyer (wgm2110),
Taimur Samee (ts2903), Rizwan Syed (rms2241)

October 16, 2017

1. Introduction

The Gantry Language is designed to make algorithmic processing of JSON data simpler. Gantry will allow for the programmatic manipulation of JSON data by implementing C-like syntax and semantics along with JSON-like[1] data types and structures.

2. Lexical Conventions

2.1 Tokens

Gantry has five types of tokens: identifiers, keywords, operators, constants, and separators.

2.1.1 Comments

Comments are lines beginning with two forward slashes, or blocks beginning with /* and ending with */ .

```
// This is a comment
```

```
/*
This is a comment
in block format
*/
```

2.1.2 Identifiers

[1] http://www.ietf.org/rfc/rfc4627.txt

An identifier, or variable name, is a sequence of alphanumeric characters or underscores that must begin with a letter. Identifiers are case-sensitive and may not be a Gantry keyword.

2.1.3 Keywords

null	int	float
string	object	bool
true	false	if
elif	else	continue
break	return	for
while		

2.1.4 Operators

<table>
<tr><th>Operator</th><th>Syntax</th><th>Operands</th></tr>
<tr><td>Arithmetic</td><td>a + b, a - b, a * b, a / b</td><td>int, float</td></tr>
<tr><td>Assignment</td><td>a = b</td><td>int, float, bool, string</td></tr>
<tr><td>Equal</td><td>a == b</td><td>int, float, bool, string</td></tr>
<tr><td>Not Equal</td><td>a != b</td><td>int, float, bool, string</td></tr>
<tr><td>Comparison</td><td>a <= b, a <b , a >= b, a >b</td><td>int, float</td></tr>
<tr><td>Logical AND</td><td>a && b</td><td>bool</td></tr>
<tr><td>Logical OR</td><td>a||b</td><td>bool</td></tr>
<tr><td>Logical NOT</td><td>!a</td><td>bool</td></tr>
<tr><td>Concatenation</td><td>aˆb</td><td>string</td></tr>
</table>

See section *3.3* for the order of operations.

2.1.5 Constants

There are four types of constants: *int*, *float*, *bool*, and *string*.

2.1.5.1 int

An int is a sequence of numeric characters [0-9] in decimal notation. They are 32-bit signed integers in the range of -2,147,483,648 to 2,147,483,647. An int must contain at least one digit.

2.1.5.2 float

Floats are real numbers with integer and decimal parts separated by a decimal point. They are 64-bit signed values in the range -3.4×10^{-38} to 3.4×10^{38}, with precision of up to 6 decimal places.

2.1.5.3 bool

Boolean values are *true* and *false*.

2.1.5.4 string

A string is an immutable sequence of zero or more ASCII characters or character escape sequences.

2.1.5.4.1 Character Escape Sequences

Character escape sequences allow for the use of certain ASCII characters in strings that overlap with language tokens as well as certain non-printable or spacing characters. The backslash character '\' signifies the beginning of a character escape sequence. The following character escape sequences are supported:

- \n yields a newline
- \r yields a carriage return
- \t yields a tab
- \b yields a backspace
- \\ yields a backslash
- \f yields a form feed
- \" yields a double-quote

3. Expressions

An expression in Gantry represents a value. Expressions consist of one or more operands and zero or more operators, where only one operator can exist between two operands. For example:

```
42
2 + 2
3 - 1
-5.0
3 / 2
```

Expressions can also be calls to functions, array subscripts, etc.

```
foo(3)
bar[2]
```

3.1 Functions

A function in Gantry must be declared in the following format:

<type> <identifier> (optional typed comma-separated list of parameters) { statements }

A function must be declared with and return a single type. A function may also include a list of typed and comma-separated parameters that will be lexically scoped into the body of the function.

Listing 1: Function Declaration

```
int repMsg(int times, string message) {
    for (int i = 0; i <= times; i++) {
        print(message ^ "\n");
    }
    return 0;
}
```

3.2 Built-In Functions

Gantry includes nine built-in functions to handle some fundamental operations that are useful for interacting with JSON-formatted data.

3.2.1 *jsonify()*

The jsonify() function takes an object as a parameter and converts the object into a JSON-formatted string. e.g.:

Listing 2: jsonify()

```
string course = "PLT";
int students = 125;
string location = "NWC";
int [] my_arr = [1,2,3]
int x = 42;
int y = 2;
object course_obj = {
    string course : course;
    int students : students;
    string location : location;
    int [] my_arr : [4, 5, 6];
    int y : x + y;
    object my_stuff : {
        string location: course_obj.location;
        string location2: location;
        int [] my_arr = my_arr;
        int [] my_arr_2 = course_obj.my_arr;
    };
```

```
};

string course_str = jsonify(course_obj);
print(course_str);
// prints {course:"PLT", students:125, location:"NWC", my_arr:[1,2,3],
// y:44, my_stuff: {location:"NWC", location2:"NWC", my_arr:[1,2,3],
// my_arr_2:[4,5,6]}}
```

3.2.2 objectify()

The objectify() function takes a string as a parameter and attempts to produce a representation of that JSON-formatted string as an object with its nested component data types. If the objectify function is passed a string that does not represent an object, the function will return { null }. e.g:

Listing 3: objectify()

```
string str = "{course:\"PLT\",students:125,location:\"NWC\"}";
object course_obj = objectify(str);
string course_name = course_obj.course;
int course_enrollment = course_obj.students;
string course_location = course_obj.location;
print(course_name);
// prints "PLT"
```

3.2.3 arrify()

The arrify() function takes a string as a parameter and attempts to produce a representation of that JSON-formatted string as an array. If the arrify function is passed a string that does not represent an array, the function will return [null]. e.g:

Listing 4: arrify()

```
string str = "[{course:\"PLT\",students:125,location:\"NWC\"},
{course:\"CS Theory\",students:200,location:\"NWC\"}]";
string [] courses_arr = arrify(str);
object first_course = courses_arr[0];
object second_course = courses_arr[1];
string first_course_name = first_course.course;
string second_course_name = second_course.course;
string output_string = first_course_name ^ " and " ^ second_course_name;
print(output_string);
// prints "PLT and CS Theory"
```

3.2.4 length()

The length() function takes an array or a string as a parameter and returns the number of elements in the array or string.

Listing 5: length()

```
string [] student_arr = ["Joe", "Bob", "Alan"];
int arr_length = length(student_arr);
print(arr_length);
// prints 3
```

3.2.5 slice()

The slice() function takes a string as a parameter with two indices. It is exclusive in that it returns a string that includes the character at the first index and it excludes the character at the second index.

Listing 6: slice()

```
string student_name = "Sandy";
string first_letter = slice(student_name, 0, 1);
print(first_letter);
// prints "S"

string new_name = "M" ^ slice(student_name, 1, 5);
print(new_name);
// prints "Mandy"

string second_new_name = "M" ^ slice(student_name, 1, 10);
print(second_new_name);
// prints "Mandy"

bool new_names_equal = (new_name == second_new_name);
print("Are the new names equal?")
if (new_names_equal) {
    print("Yes")
} else {
    print("No")
}
print("Are the new names equal? " ^ new_names_equal);
// prints "Are the new names equal? Yes"

print("Old name : " ^ student_name ^ " New name : " ^ new_name);
// prints "Old name: Sandy New name: Mandy"
```

3.2.6 print()

The print() function takes a parameter of any type defined in our language and print its string representation.

Listing 7: print()

```
string course_name = "PLT";
print("This is the course name: "^ course_name);
// prints "This is the course name : PLT"
```

3.2.7 to_string()

The to_string() function takes a parameter of any type defined in our language and returns it as a string.

Listing 8: to_string()

```
int course_enrollment = 3;
string course_enrollment_string = to_string(course_enrollment)
print(course_enrollment_string);
// prints 3
```

3.2.8 http_get()

The http_get() function takes a server and port as a parameter along with a URI, and sends an HTTP GET request.

Listing 9: http_get()

```
/*
  Returns a json object of containers running on
  a particular Docker engine.
*/
string uri = "/v1.19/containers/json";
string cons = http_get("192.168.0.9", 80, uri);
object [] cons_arr = arrify(cons);
print(cons_arr);
```

3.2.9 http_post()

The http_post() function takes a server, port, URI, and POST data as parameters to form an HTTP POST request.

Listing 10: http_post()

```
/*
  Returns a json object of a newly created container
  running on a particular Docker engine.
*/
string post_data = "{"Image": "centos", "Cmd": ["echo", "hello world"]}";
string uri = "/v1.19/containers/create";
string con = http_post("192.168.0.9", 80, uri, post_data);
object con_obj = objectify(con);
print(con_obj);
```

3.3 Operator Precedence

The following table lists the operator precedence. Operators with a lower numeric value are considered higher priority.

Precedence	Operand	Description	Associativity
1	() [] . ++ --	Parentheses Brackets(array access) Member selection Postfix increment/decrement	Left-to-right
2	+ - !	Unary plus/minus Logical negation	Right-to-left
3	* /	Multiplication Division	Left-to-right
4	+ -	Addition, Subtraction	Right-to-left
5	< <= >= >	Relational less-than/or equal to Relational greater-than/or equal to	Left-to-right
6	^	String Concatenation	Left-to-right
7	== !=	Relational Equality Operators	Left-to-right
8	&&	Logical AND	Left-to-right
9	||	Logical OR	Left-to-right
10	=	Assignment	Right-to-left
11	,	Comma for Next Argument	Left-to-right

4. Statements

A statement in Gantry performs an action such as evaluation or control-flow. A statement may also contain expressions.

4.1 Expression-Statements

While statements differ from expressions in that an expression represents a value and a statement performs an action, we can combine these two concepts syntactically by adding a succeeding semi-colon to any expression. This produces an expression-statement wherein the value represented by the expression is evaluated *only* because it is also a statement.

Listing 11: Expression-Statements

```
42;
2 + 2;
3 - 1;
foo();
bar();
```

4.2 Control-Statements

Note that a conditional containing a type other than a boolean will evaluate to *true* only if it is not empty or non-zero. e.g. a non-zero integer or float, a not empty string, a not empty array, or a not empty object.

Listing 12: If-Statement

```
if (value) {
    print(value);
}
elif (value_2) {
    print(value_2);
}
else {
    print(value_3);
}
```

Listing 13: While-Loop

```
while (value) {
    print(value);
}
```

Listing 14: For-Loop

```
for (int i = 0; i <= 3; i++) {
    print(i);
}
```

4.3 Jump statements

Jump statements cause unconditional jumps to other parts of the code, allowing for the transfer of control to other parts of the program.

4.3.1 *continue*

Continue statements pass control back to the enclosing conditional *while* or *for* statement.

Listing 15: continue

```
while(x < 4) {
    continue;
    x++
}
```

Note that the code underneath the continue statement is never executed, so the loop carries on forever.

4.3.2 *break*

Break statements terminate the execution of the enclosing *while* or *for* loop. Control then passes to the succeeding statement outside of the loop body.

Listing 16: break

```
while(x < 4) {
    break;
    x++
}
```

Unlike in the example for *continue*, the loop terminates at the *break* statement. The variable still does not increment, but there is not an infinite loop, as the loop ends as soon as *break* is executed.

4.3.3 *return*

Return statements end the current function and return control to the caller. Any number of return statements are allowed in a function, but each *return* must ony return a single value that matches the return type of the function it is within. Note that a function of return type *null* will not support statements that *return* a value.

Listing 17: return

```
boolean isHeader(string s) {
    if(s) {
        return true;
    } else {
        return false;
    }
}
```

4.4 ***Comparison Operators***

4.4.1 *Equality Operators*

There are two equality operators == and ! = which can be used to evaluate the equality of the *content* of two operands. Such operands must be of the same type, where valid types are *int*, *float*, *bool*, and *string*. The equality evaluation will return a *boolean* value of either *true* or *false*.

4.4.2 *Relational Operators*

There are four relational operators <, >, <=, and >= which can be used to compare two operands. Such operands must be of the same type, where valid types are *int* and *float*. The relational evaluation will return a boolean value of either *true* or *false*.

4.4.3 *Logical Operators*

There are three logical operators && (AND), || (OR), and ! (NOT), where AND and OR evaluate two operands, and NOT evaluates a single operand. All operands must be of type *bool*. The logical evaluation will return a boolean value of either *true* or *false*.

4.7 ***Assignment Expressions***

An assignment expression assigns a value to an identifier. An assignment expression must include a type and a value to which the identifier will be initialized.

Valid types are *bool, int, float, string, array*, and *object*. Note that an *object* is a composite type and an *array* is an aggregate type with a special declaration syntax outlined in section *4.7.2*.

4.7.1 *Identifiers*

Identifiers must be declared and initialized in the following format:

<type> <identifier> = value of *type*;

Listing 18: Identifier Declarations

```
int y = 42;
// initializes an integer named y with a value of 42
```

See section *4.7* for valid types.

4.7.2 *Arrays*

Arrays must be declared and initialized in the following format:

<type> [ ] <identifier> = [comma-separated values of *type*]

Listing 19: Array Declarations

```
int [] exampleArray2 = [1,10,100];
// initializes an array of integers
```

Subscripts may be used to access or modify individual elements of an array. A subscript may consist of any expression that evaluates to an integer, as long as the integer is within the bounds of the array. Array indices start at 0.

Listing 20: Array Subscripting

```
int [] exampleArray2 = [1,10,100];
int val2 = exampleArray2[1];
// val2 is 10
exampleArray2[1] = 20;
```

See section *4.7* for valid types.

4.7.3 Objects

Objects must be declared and initialized in the following format:

Listing 21: Object Declarations

```
int x = 1;
object v = { int i: 1, int j: x, string j: "hello world" }
// initializes an object with two integers and a string
```

Object dot notation can be used to access or modify the value of a key that is a member of an Object. Dot notation can also be chained if there are nested objects.

Listing 22: Object Dot Notation

```
object v = { int i: 1, int j: x, string j: "hello world" }
int j = v.i;
// value of j = 1
```

See section *4.7* for valid types.

5. Grammar

Terminals are in *italics*.

program:
  declaration-list$_{opt}$ ***eof***

declaration-list
  declaration
  declaration-list declaration

declaration
  statement
  function-declaration

type-specifier:
  int
  float
  object
  string
  bool

null

statement-list:
statement
statement-list ; statement

statement:
for-statement
if-statement
while-statement
jump-statement
expression-statement

function-parameter:
type-specifier *identifier*

function-parameter-list:
function-parameter
function-parameter-list, function-parameter

function-declaration:
type-specifier *identifier* (function-parameter-list$_{opt}$) { statement-list }
type-specifier [ ] *identifier* (function-parameter-list$_{opt}$) { statement-list }

function-expression:
identifier (expression-list$_{opt}$)

expression:
identifier
constant
array-expression
object-expression
arithmetic-expression
comparison-expression
logical-expression
assignment-expression
string-concat-expression

arithmetic-expression:
expression + expression
expression − expression
expression ∗ expression
expression / expression

expression ++
expression --

comparison-expression:
expression < expression
expression > expression
expression <= expression
expression >= expression
expression == expression
expression ! = expression

logical-expression:
expression && expression
expression || expression
!expression

string-concat-expression:
expression ^ expression

assignment-expression:
identifier = expression
type-specifier *identifier* = expression
identifier [ ] = expression
type-specifier [ ] *identifier* = expression

expression-statement:
expression ;
assignment-expression ;
function-expression ;

for-statement:
for (expression ; expression ; expression) { statement-list }

if-statement:
if (expression) { statement-list }
if (expression) { statement-list } else { statement-list }
if (expression) { statement-list } elif (expression) { statement-list } else { statement-list }

while-statement:
while (expression) { statement-list }

jump-statement:

break ;
continue ;
return expression ;

object-expression:
{ key-value-list$_{opt}$ }

key-value-list-opt:
key-value-list

key-value-list:
key-value
key-value-list, key-value

key-value:
type-specifier *identifier* : expression

array-expression:
[expression-list$_{opt}$]

expression-list:
expression
expression-list , expression

expression-list-opt:
expression-list

object-expression-list:
object-expression
object-expression-list, object-expression

identifier-list:
identifier
identifier-list, *identifier*

constant-list:
constant
constant-list, constant

constant:
true
false
null

literal

literal:

int-literal
float-literal
string-literal