LQX Users Guide

General Purpose Language for Simulation Modeling

Tuesday, 3 February 2009 - Martin Mroz

Contents

1 Introduction to LQX

The LQX programming language is a general purpose programming language used for the control of input parameters to the Layer Queueing Network Solversystem for the purposes of sensitivity analysis. This language allows a user to perform a wide range of different actions on a variety of different input sources, and to subsequently solve the model and control the output of the resulting data.

1.1 Input File Format

The LQX programming language follows grammar rules which are very similar to those of ANSI C and PHP. The main difference between these languages and LQX is that LQX is a loosely typed language with strict runtime type-checking and a lack of variable coercion ("type casting"). Additionally, variables need not be declared before their first use. They do, however, have to be initialized. If they are un-initialized prior to their first use, the program will fail.

1.1.1 Comment Style

LQX supports two of the most common commenting syntaxes, "C-style" and "C++-style." Any time the scanner discovers two forward slashes side-by-side (`//`), it skips any remaining text on that line (until it reaches a newline). These are "C++-style" comments. The other rule that the scanner uses is that should it encounter a forward slash followed by an asterisk ("/*"), it will ignore any text it finds up until a terminating asterisk followed by a slash ("*/"). The preferred commenting style in LQX programs is to use "C++-style" comments for single-line comments and to use "C-style" comments where they span multiple lines. This is a matter of style.

1.1.2 Intrinsic Types

There are 5 intrinsic types in the LQX programming languages:

- **Number**: All numbers are stored in IEEE double-precision floating point format.
- **String**: Any literal values between (") and (") in the input.
- **Null**: This is a special type used to refer to an "empty" variable.
- **Boolean**: A type whose value is limited to either "true" or "false."
- **Object**: An semi-opaque type used for storing complex objects. See "Objects."
- **File Handle** File handles to open files for writing/appending or reading. See "File Handles."

LQX also supports a pseudo-intrinsic "Array" type. Whereas for any other object types, the only way to interact with them is to explicitly invoke a method on them, objects of type Array may be accessed with `operator []` and with `operator []=`, in a familiar C- and C++-style syntax.

The Object type also allows certain attributes to be exposed as "properties." These values are accessed with the traditional C-style `object.property` syntax. An example property is the `size` property for an object of type Array, accessed as `array.size` Only instances of type Object or its derivatives have properties. Number, String, Null and Boolean instances all have no properties.

1.1.3 Arrays and Iteration

The built-in Array type is very similar to that used by PHP. It is actually a hash table, also known as a "Dictionary" or a "Map" for which you may use any object as a key, and any object as a value. It is important to realize that different types of keys will reference different entries. That is to say that `integer 0` and `string ''0''` will not yield the same value from the Array when used as a key.

The Array object exposes a couple of convenience APIs, as detailed in Appendix A. These methods are simply short-hand notation for the full function calls they replace, and provide no additional functionality. Arrays may be created in three different ways:

- `array_create(...)` and `array_create_map(key,value,...)`:
  The explicit, but long and wordy way of creating an array of objects or a map is by using the standard functional API. `array_create(...)` takes an arbitrary number of parameters (from 0 up to the maximum specified, for all practical purposes infinity), and returns a new Array instance consisting of `[0=>arg1, 1=>arg2, 2=>arg3, ...]`.

  The other function, `array_create_map(key,value,...)` takes an even number of arguments, from 0 to 2n. The first argument is used as the key, and the second argument used as the value for that key, and so on. The resulting Array instance consists of `[arg1=>arg2, arg3=>arg4, ...]`. Both of these methods are documented in Appendix A.
- [`arg1, arg2, ...`]: Shorthand notation for `array_create(...)`
- {`k1=>v1, k2=>v2, ...`}: Shorthand notation for `array_create_map(...)`

The LQX language supports two different methods of iterating over the contents of an Array. The first involves knowing what the keys in the array actually are. This is a "traditional" iteration.

```
1 /* Traditional Array Iteration */
2 for (idx = 0; key < array.size; idx=idx+1) {
3   print("Key ", idx, " => ", array[idx]);
4 }
```

In the above code snippet, we assume there exists an array which contains `n` values, stored at indexes 0 through `n-1`, continuously. However, the language provides a more elegant method for iterating over the contents of an array which does not require prior knowledge of the contents of the array. This is known as a "`foreach`" loop. The statement above can be rewritten as follows:

```
1 /* More modern array itteration */
2 foreach (key, value in array) {
3   print("Key ", key, " => ", value);
4 }
```

This method of iteration is much cleaner and is the recommended way of iterating over the contents of an array. However, there is little guarantee of the order of the results in a `foreach` loop, especially when keys of multiple different types are used.

1.1.4 Type Casting

The LQX programming language provides a number of built-in methods for converting between variables of different types. Any of these methods support any input value type except for the Object type. The following is a non-extensive list of use cases for each of the different type casting methods and the results. Complete documentation is provided in Appendix A.

str(...)	
`str()`	“”
`str(1.0)`	“1”
`str(1.0, "+", true)`	“1+true”
`str([1.0, "t"])`	“[0=>1, 1=>t]”
`str(null)`	“(null)”

double(?)	
`double(1.0)`	1.0
`double(null)`	0.0
`double("9")`	9.0
`double(true)`	1.0
`double([0])`	`null`

boolean(?)	
`boolean(1.0)`	`true`
`boolean(17.0)`	`true`
`boolean(-9.0)`	`true`
`boolean(0.0)`	`false`
`boolean(null)`	`false`
`boolean("yes")`	`true`
`boolean(true)`	`true`
`boolean([0])`	`null`

1.1.5 User-Defined Functions

The LQX programming language has support for user-defined functions. When defined in the language, functions do not check their arguments types so every effort must be taken to ensure that arguments are the type that you expect them to be. The number of arguments will be checked. Variable-length argument lists are also supported with the use of the ellipsis (`...`) notation. Any arguments given that fall into the ellipsis are converted into an array named (`_va_list`) in the functions' scope. This is a regular instance of Array consisting of 0 or more items and can be operated on using any of the standard operators.

User-defined functions do **not** have access to any variables except their arguments and External ($-prefixed) and Constant (@-prefixed) variables. Any additional variables must be passed in as arguments, and all values must be returned. All arguments are in **only**. There are no out or inout arguments supported. All arguments are copied, pass-by-value. The basic syntax for declaring functions is as follows:

```
function <name>(<arg1>, <arg2>, ...) {
  <body>
  return (value);
}
```

You can return a value from a function anywhere in the body using the `return` function. A function which reaches the end of its body without a call to return will automatically return NULL. `return()` is a function, not a language construct, and as such the brackets are required. The number of arguments is not limited, so long as each one has a unique name there are no other constraints.

1.2 Writing Programs in LQX

1.2.1 Hello, World Program

A good place to start learning how to write programs in LQX is of course the traditional Hello World program. This would actually be a single line, and is not particularly interesting. This would be as follows:

```
1 println("Hello, World!");
```

The "`println()`" function takes an arbitrary number of arguments of any type and will output them (barring a file handle as the first parameter) to standard output, followed by a newline.

1.2.2 Fibonacci Sequence

This particular program is a great example of how to perform flow control using the LQX programming language. The Fibonacci sequence is an extremely simple infinite sequence which is defined as the following piecewise function:

$$\text{fib}(X) = \begin{cases} 1 & x = 0, 1 \\ \text{fib}(x-1) + \text{fib}(x-2) & \text{otherwise} \end{cases} \tag{1}$$

Thus we can see that the Fibonacci sequence is defined as a recursive sequence. The naive approach would be to write this code as a recursive function. However, this is extremely inefficient as the overhead of even simple recursion in LQX can be substantial. The best way is to roll the algorithm into into a loop of some type. In this case, the loop is terminated when we have reached a target number in the Fibonacci sequence { 1, 1, 2, 3, 5, 8, 13, 21, ...}.

```
1  /* Initial Values */
2  fib_n_minus_two = 1;
3  fib_n_minus_one = 1;
4  fib_n = 0;
5
6  /* Loop until we reach 21 */
7  while (fib_n < 21) {
8    fib_n = fib_n_minus_one + fib_n_minus_two;
9    fib_n_minus_two = fib_n_minus_one;
10   fib_n_minus_one = fib_n;
11   println("Currently: ", fib_n);
12 }
```

As you can see, this language is extremely similar to C or PHP. One of the few differences as far as expressions are concerned is that pre-increment/decrement and post-increment/decrement are not supported. Neither are short form expressions such as `+=, -=, *=, /=`, etc.

1.2.3 Re-using Code Sections

Many times, there will be code in your LQX programs that you would like to invoke in many places, varying only the parameters. The LQX programming language does provide a pretty standard functions system as described earlier. Bearing in mind the caveats (some degree of overhead in function calls, plus the inability to see global variables without having them passed in), we can make pretty ingenious use of user-defined functions within LQX code.

When defining functions, you can specify only the number of arguments, not their types, so you need to make sure things are what you expect them to be, or your code may not perform as you expect. We will begin by demonstrating a substantially shorter (but as described earlier) much less efficient implementation of the Fibonacci Sequence using functions and recursion.

```
function fib(n) {
  if (n == 0 || n == 1) { return (1); }
  return (fib(n-2) + fib(n-1));
}
```

Once defined, a function may be used anywhere in your code, even in other user defined functions (and itself — recursively). This particular example functions very well for the first 10-11 fibonacci numbers but becomes substantially slower due to the increased number of relatively expensive function invocations. *Remember*, `return()` is a function, not a language construct. The brackets are required.

A much more interesting use of functions, specifically those with variable length argument lists, is an implementation of the formula for standard deviation of a set of values:

```
function average(/*Array<double>*/ inputs) {
  double sum = 0.0;
  foreach (v in inputs) { sum = sum + v; }
  return (sum / inputs.size);
}

function stdev(/*boolean*/ sample, ...) {
  x_bar = average(_va_list);
  sum_of_diff = 0.0;

  /* Figure out the divisor */
  divisor = _va_list.size;
  if (sample == true) {
    divisor = divisor - 1;
  }

  /* Compute sum of difference */
  foreach (v in _va_list) {
    sum_of_diff = sum_of_diff + pow(v - x_bar, 2);
  }

  return (pow(sum_of_diff / divisor, 0.5));
}
```

You can then proceed to compute the standard deviation of the variable length of arguments for either sample or non-sample values as follows, from anywhere in your program after it has been defined:

```
stdev(true,  1, 2, 5, 7, 9, 11);
stdev(false, 2, 9, 3, 4, 2);
```

1.2.4 Using and Iterating over Arrays

As mentioned in the "Arrays and Iteration" under section 1.1 of the Manual, LQX supports intrinsic arrays and `foreach` iteration. Additionally, any type of object may be used as either a key or a value in the array. The following example illustrates how values may be added to an array, and how you can iterate over its contents and print it out. The following snippet creates an array, stores some key-value pairs with different types of keys and values, looks up a couple of them and then iterates over all of them.

```
1  /* Create an Array */
2  array = array\_create();
3
4  /* Store some key-value pairs */
5  array[0] = "Slappy";
6  array[1] = "Skippy";
7  array[2] = "Jimmy";
8
9  /* Iterate over the names */
10 foreach ( index,name in array ) {
11   print("Chipmunk #", index, " = ", name);
12 }
13
14 /* Store variables of different types, shorthand */
15 array = {true => 1.0, false => 3.0, "one" => true, "three" => false}
16
17 /* Shorthand indexed creation with iteration */
18 foreach (value in [1,1,2,3,5,8,13]) {
19   print ("Next fibonacci is ", value);
20 }
```

1.3 Program Input/Output and External Control

The LQX language allows users to write formatted output to external files and standard output and to read input data from external files/pipes and standard input. These features may be combined to allow LQNX to be controlled by a parent process as a child process providing model solving functionality. These capabilities will be described in the following sections.

1.3.1 File Handles

The LQX language allows users to open files for program input and output. Handles to these open files are stored in the symbol table for use by the print() functions for file output and the read_data() function for data input. Files may be opened for writing/appending or for reading. The LQX interpreter keeps track of which file handles were opened for writing and which were opened for reading.

The following command opens a file for writing. If it exists it is overwritten. It is also possible to append to an existing file. The three options for the third parameter are `write`, `append`, and `read`.

```
file_open( output_file1, "test_output_99-peva.txt", write );
```

To close an open file handle the following command is used.

```
file_close( output_file1 );
```

1.3.2 File Output

Program output to both files and standard output is possible with the print functions. If the first parameter to the functions is an existing file handle opened for writing output is directed to that file. If the first parameter is not a file handle output is sent to standard output. Standard output is useful when it is desired to control LQNX execution from a parent process using pipes. If the given file handle has been opened for reading instead of writing a runtime error results.

There are four variations of print commands with two options. One option is a newline at the end of the line. It is possible to specify additional newlines with the `endl` parameter. The second option is controlling the spacing between columns either by specifying column widths in integers or supplying a text string to be placed between columns.

The basic print functions are `print()` and `println()` with the `ln` specifying a newline at the end.

```
println( output_file1, "Model run #:  ", i, " t1.throughput:  ", t1.throughput );

print( output_file1, "Model run #:  ", i, " t1.throughput:  ", t1.throughput, endl );
```

It should be noted that with the extra `endl` parameter both of these calls will produce the same output. The acceptable inputs to all print functions are valid file handles, quoted strings, LQX variables that evaluate to numerical or boolean values (or expressions that evaluate to numerical/boolean values) as well as the newline specifier `endl`. Parameters should be separated by commas.

To print to standard output no file handle is specified as follows:

```
println( "subprocess lqns run #:  ", i, " t1.throughput:  ", t1.throughput );
```

To specify the content between columns the print functions `print_spaced()` and `println_spaced()` are used. The first parameter after the file handle (the second parameter when a file handle is specified) is used to specify either column widths or a text string to be placed between columns. If no file handle is specified as when printing to standard output then the first parameter is expected to be the spacing specifier. The specifier must be either an integer or a string.

The following `println_spaced()` command specifies the string `", "` to be placed between columns. It could be used to create comma separated value (csv) files.

```
println_spaced( output_file2, ", ", $p1, $p2, $y1, $y2, t1.throughput );
```

Example output: 0, 2, 0.1, 0.05, 0.0907554

The following `println_spaced()` command specifies the integer 12 as the column width.

```
println_spaced( output_file3, 12, $p1, $p2, $y1, $y2, t1.throughput );
```

1.3.3 Reading Input Data from Files/Pipes

Reading data from input files/pipes is done with the `read_data()` function. Data can either be read from a valid file handle that has been opened for reading or from standard input. Reading data from standard input is useful when is useful when it is desired to control LQNX execution from a parent process using pipes. If the given file handle has been opened for writing rather than reading a runtime error results. The first parameter is either a valid file handle for reading or the strings `stdout` or `-` specifying standard input. The data that can be read can be either numerical values or boolean values.

There are two forms in which the `read_data()` function can be used. The first is by specifying a list of LQX variables which correspond to the expected inputs from the file/pipe. This requires the data inputs from the pipe to be in the expected order.

```
read_data( input_file, y, p, keep_running );
```

The second form in which the `read_data()` function can be used is much more robust. It can go into a loop attempting to read string/value pairs from the input pipe until a termination string `STOP_READ` is encountered. The string must corespond to an existing LQX variable (either numeric or boolean) and the corresponding value must be of the same type.

```
read_data( stdin, read_loop );
```

Sample input:

y 10.0 p 1.0 STOP_READ

continue_processing false STOP_READ

1.3.4 Controlling LQNX from a Parent Process

The file output and data reading functions can be combined to allow an LQNX process to be created and controlled by a parent process through pipes. Input data can be read in from pipes, be used to solve a model with those parameters and the output of the solve can be sent back through the pipes to the parent process for analysis. A LQX program can easily be written to contain a main loop that reads input, solves the model, and returns output for analysis. The termination of the loop can be controlled by a boolean flag that can be set from the parent process.

This section describes an example of how to control LQNX execution from a parent process, in this case a `perl` script which uses the `open2()` function to create a child process with both the standard input and output mapped to file handles in the `perl` parent process. This allows data sent from the parent to be read with read_data(stdin, ...) and output from the LQX print statements sent to standard output to be received for analysis in the parent.

This also provides synchronization between the parent and the child LQNX processes. The read_data() function blocks the LQNX process until it has received its expected data. Similarly the parent process can be programmed to wait for feedback from the child LQNX process before it continues.

The following is an example perl script that can be used to control a LQNX child process.

```
1  #!/usr/bin/perl -w
2  # script to test the creation and control of an lqns solver subprocess
3  # using the LQX language with synchronization
4
5  use FileHandle;
6  use IPC::Open2;
7
8  @phases = ( 0.0, 0.25, 0.5, 0.75, 1.0 );
9  @calls = ( 0.1, 3.0, 10.0 );
10
11 # run lqnx as subprocess receiving data from standard input
12 open2( *lqnxOutput, *lqnxInput, "lqnx 99-peva-pipe.lqnx" );
13
14 for $call (@calls) {
15   for $phase (@phases) {
```

```
16          print( lqnxInput "y ", $call, " p ", $phase, " STOP_READ " );
17          while( $response = <lqnxOutput>) !~ m/subprocess lqns run/ ){}
18          print( "Response from lqnx subprocess: ", $response );
19        }
20      }
21
22      # send data to terminate lqnx process
23      print( lqnxInput "continue_processing false STOP_READ" );
```

The above program invokes the lqnx program with its input file as a child process with `open2()`. Two file handles are passed as parameters. These will be used to send data over the pipe to the LQNX process to be received as standard input and to receive feedback from the LQX program which it sends as standard output.

The while loop at line 17 waits for the desired feedback from the model solve before continuing. This example uses stored data but a real application such as optimization would need to analyze the feedback data to decide which data to send back in the next iteration therefore this synchronization is important.

When the data is exhausted the LQNX process needs to be told to quit. This is done with the final print statement which sets the continue_processing flag to false. This causes the main loop in the LQX program which follows to quit.

```
1   <lqx><![CDATA[
2
3       i = 1;
4       p = 0.0;
5       y = 0.0;
6       continue_processing = true;
7
8       while ( continue_processing ) {
9
10        read_data( stdin, read_loop ); /* read data from input pipe */
11
12        if( continue_processing ) {
13
14          $p1 = 2.0 * p;
15          $p2 = 2.0 * (1 - p);
16          $y1 = y;
17          $y2 = 0.5 * y;
18          solve();
19
20          /* send output of solve through stdout through pipe */
21          println( "subprocess lqns run #: ", i, " t1.throughput: ", t1.throughput );
22          i = i + 1;
23        }
24      }
25  ]]></lqx>
```

The variables `p`, `y`, and `continue_processing` all need to be initialized to their correct types before the loop begins as they need to exist when the `read_data()` function searches for them in the symbol table. This is necessary as they are all local variables. External variables that exist in the LQN model such as `$p` and `$y` don't need initialization.

1.4 Actual Example of an LQX Model Program

The following LQX code is the complete LQX program for the model designated `peva-99`. The model itself contains a few model parameters which the LQX code configures, notably `$p1`, `$p2`, `$y1` and `$y2`. The LQX program is responsible for setting the values of all model parameters at least once, invoking solve and optionally printing out certain result values. Accessing of result values is done via the LQNS bindings API documented in Section 3.

The program begins by defining an array of values that it will be setting for each of the external variables. By enumerating as follows, the program will set the variables for the cross product of `phase` and `calls`.

```
1 phase = [ 0.0, 0.25, 0.5, 0.75, 1.0 ];
2 calls = [ 0.1, 3.0, 10.0 ];
3 foreach ( idx,p in phase ) {
4   foreach ( idx,y in calls ) {
```

Next, the program uses the input values `p` and `y` to compute the values of `$p1`, `$p2`, `$y1` and `$y2`. Any assignment to a variable beginning with a `$` requires that variable to have been defined externally, within the model definition. When such an assignment is made the value of the right-hand side is effectively put everywhere the left-hand side is found within the model.

```
5       $p1 = 2.0 * p;
6       $p2 = 2.0 * (1 - p);
7       $y1 = y;
8       $y2 = 0.5 * y;
```

Since all variables have now been set, the program invokes the solve function with its optional parameter, the suffix to use for the output file of the current run. This particular program outputs `in.out-$p1-$p2-$y1-$y2` files, so that results for a given set of input values can easily be found. As shown in the documentation in Section 3, `solve(<opt> suffix)` will return a boolean indicating whether or not the solution converged, and this program will abort when that happens, although that is certainly not a requirement.

```
9       if (solve(str($p1,"-",$p2,"-",$y1,"-",$y2)) == false) {
10        println("peva-99.xml:LQX: Failed to solve the model properly.");
11        abort(1, "Failed to solve the model.");
12      } else {
```

The remainder of the program outputs a small table of results for certain key values of interest to the person running the solution using the APIs in Section 3.

```
13        t0 = task("t0");
14        p0 = processor("p0");
15        e0 = entry("e0");
16        ph1 = phase(e0, 1);
17        ctoe1 = call(ph1, "e1");
18        println("+-------------------------------------+");
19        println("t0 Throughput:  ", t0.throughput        );
20        println("t0 Utilization: ", t0.utilization       );
21        println("+                -----                +");
22        println("e0 Throughput:  ", e0.throughput        );
23        println("e0 TP Bound:    ", e0.throughput_bound  );
24        println("e0 Utilization: ", e0.utilization       );
25        println("+                -----                +");
26        println("ph Utilization: ", ph1.utilization      );
27        println("ph Svt Variance:", ph1.service_time_v   );
28        println("ph Service Time:", ph1.service_time     );
```

```
29        println("ph Proc Waiting:", ph1.proc_waiting      );
30        println("+                 -----                 +");
31        println("call Wait Time: ", ctoe1.wait_time       );
32        println("+-------------------------------------+");
33      }
34    }
35  }
```

2 API Documentation

2.1 Built-in Class: Array

Summary of Attributes		
numeric	`size`	The number of key-value pairs stored in the array.

Summary of Constructors		
object[Array]	`array_create(...)`	This method returns a new instance of the Array class, where each the first argument to the method is mapped to index numeric(0), the second one to numeric(1) and so on, yielding `[0=>arg0, 1=>arg1, ...]`
object[Array]	`array_create_map(k,v,...)`	This method returns a new instance of the Array class where the first argument to the constructor is used as the key, and the second is used as the value, and so on. The result is a n array `[arg0=>arg1, arg2=>arg3,...]`

Summary of Methods		
null	`array_set(object[Array] a, ? key, ? value)`	This method sets the value `value` of any type for the key `key` of any type, for array `a`. The shorthand notation for this operation is to use the `operator []`.
ref<?>	`array_get(object[Array] a, ? key)`	This method obtains a reference to the slot in the array `a` for the key `key`. If there is no value defined in the array yet for the given key, a new slot is created for that key, assigned to NULL, and a reference returned.
boolean	`array_has(object[Array] a, ? key)`	Returns whether or not there is a value defined on array `a` for the given key, `key`.

2.2 Built-in Global Methods and Constants

2.2.1 Intrinsic Constants

Summary of Constants		
double	`@infinity`	IEEE floating-point numeric infinity.
double	`@type_un`	The type_id for an Undefined Variable.
double	`@type_boolean`	The type_id for a Boolean Variable.
double	`@type_double`	The type_id for a Numeric Variable.
double	`@type_string`	The type_id for a String Variable.
double	`@type_null`	The type_id for a Null Variable.

2.2.2 General Utility Functions

Summary of Methods		
null	`abort(numeric n, string r)`	This call will immediately halt the flow of the program, with failure code `n` and description string `r`. This cannot be "caught" in any way by the program and will result in the interpreter not executing any more of the program.
null	`copyright()`	Displays the LQX copyright message.
null	`print_symbol_table()`	This is a very useful debugging tool which output the name and value of all variables in the current interpreter scope.
null	`print_special_table()`	This is also a useful debugging tool which outputs the name and value of all special (External and Constant) variables in the interpreter scope.
numeric	`type_id(?  any)`	This method returns the Type ID of any variable, including intrinsic types (numeric, boolean, null, etc.) and the result can be matched to the constants prefixed with @type (@type_null, @type_un, @type_double, etc.)
null	`return(?  any)`	This method will return any value from a user-defined function. This method cannot be used in global scope.

2.2.3 Numeric/Floating-Point Utility Functions

Summary of Methods		
numeric	`abs(numeric n)`	Returns the absolute value of the argument `n`
numeric	`ceil(numeric n)`	Returns the value of `n` rounded up.
numeric	`floor(numeric n)`	Returns the value of `n` rounded down.
numeric	`pow(numeric bas, numeric x)`	Returns `bas` to the power `x`.

2.2.4 Type-casting Functions

Summary of Methods		
string	`str(...)`	This method will return the same value as the function `print(...)` would have displayed on the screen. Each argument is coerced to a string and then adjacent values are concatenated.
numeric	`double(?  x)`	This method will return 1.0 or 0.0 if provided a boolean of `true` or `false` respectively. It will return the passed value for a double, 0.0 for a null and fail (NULL) for an object. If it was passed a string, it will attempt to convert it to a double. If the whole string was not numeric, it will return NULL, otherwise it will return the decoded numeric value.
boolean	`bool(?  x)`	This method will return `true` for a numeric value of (not 0.0), a boolean `true` or a string "true" or "yes". It will return `false` for a numeric value 0.0, a NULL or a string "false" or "no", or a boolean `false`. It will return NULL otherwise.

3 API Documentation for the LQNS Bindings

3.1 LQNS2 Class: Processor

Summary of Attributes		
double	`utilization`	The utilization of the Processor

Summary of Constructors		
Processor	`processor(string name)`	Returns an instance of Processor from the current LQNS2 model with the given name.

3.2 LQNS2 Class: Task

Summary of Attributes		
double	`throughput`	The throughput of the Task
double	`utilization`	The utilization of the Task
double	`proc_utilization`	This Task's processor utilization
Array	`phase_utilizations`	Individual phase utilizations

Summary of Constructors		
Task	`task(string name)`	Returns an instance of Task from the current LQNS2 model with the given name.

3.3 LQNS2 Class: Entry

Summary of Attributes		
double	`throughput`	Entry throughput
double	`throughput_bound`	Entry throughput bound
double	`utilization`	Entry utilization
double	`proc_utilization`	Entry processor utilization
double	`coeff_variation_sq`	Squared coefficient of variation
double	`open_wait_time`	Entry open wait time
boolean	`has_phase_1`	Whether the entry has a phase 1 result
boolean	`has_phase_2`	Whether the entry has a phase 2 result
boolean	`has_open_wait_time`	Whether the entry has an open wait time
double	`ph_1_service_time`	Phase 1 Service Time
double	`ph_1_service_time_v`	Phase 1 Service Time Variance
double	`ph_1_proc_waiting`	Phase 1 Processor Wait Time
double	`ph_2_service_time`	Phase 2 Service Time
double	`ph_2_service_time_v`	Phase 2 Service Time Variance
double	`ph_2_proc_waiting`	Phase 2 Processor Wait Time

Summary of Constructors		
Entry	`entry(string name)`	Returns the Entry object for the model entry whose name is given as name

3.4 LQNS2 Class: Phase

Summary of Attributes		
double	service_time	Phase service time
double	service_time_v	Phase service time variance
double	utilization	Phase utilization
double	proc_waiting	Phases' processor waiting time

Summary of Constructors		
Phase	phase(object entry, numeric_int nr)	Returns the Phase object for a given entry's phase number specified as nr

3.5 LQNS2 Class: Call

Summary of Attributes		
double	wait_time	Call waiting time

Summary of Constructors		
Call	call(object phase, string destinationEntry)	Returns the call from an entry's phase (phase) to the destination entry whose name is (dest)

3.6 LQNS2 Class: Activity

Summary of Attributes		
double	service_time	Activity service time
double	service_time_v	Activity service time variance
double	utilization	Activity utilization
double	proc_waiting	Activities' processor waiting time
double	cv_squared	The square of the coefficient of variation
double	throughput	The activity throughput
double	proc_utilization	The activities' share of the processor utilization

Summary of Constructors		
Activity	activity(object task, string name)	Returns an instance of Activity from the current LQNS2 model, whose name corresponds to an activity in the given task.