NetXMS Scripting Language

Table of Contents

Introduction

In many parts of the system, fine tuning can be done by using NetXMS built-in scripting language called NXSL (stands for NetXMS Scripting Language). NXSL was designed specifically to be used as embedded scripting language within NetXMS, and because of this has some specific features and limitations. Most notable is very limited access to data outside script boundaries – for example, from NXSL script you cannot access files on server, nor call external programs, nor even access data of the node object other than script is running for without explicit permission. NXSL is interpreted language – scripts first compiled into internal representation (similar to byte code in Java), which than executed inside NXSL VM.

Script security

Because NXSL provides functions for searching objects, and because all scripts are executed on management server, user with write access to only one node can potentially acquire information about nodes to which he normally does not have access. For example, without additional security checks user with write access to node `A` and no access to node `B` can create transformation script for DCI on node `A` and use FindNodeObject function to access node `B` and get information about it, thus breaking security settings.

To prevent such scenario, all NXSL functions capable of accessing NetXMS objects requires "current node" object to be provided. Reference to object being searched will only be returned if node object supplied as "current node" is in trusted nodes list of target object. For example, if variable `$node` in script refers to `NODE1`, and `FindNodeObject($node, "NODE2")` called, `NODE1` must be added to list of trusted nodes for `NODE2`. In most places (transformation script, event processing policy, etc.) predefined variable `$node` exists, which refers to node object on behalf of which script is being executed. It will be event source for event processing policy script, DCI owner for transformation script, and so on.

For environments where such strict security checks are not required (for example, all users have read access to all nodes), they can be disabled to simplify configuration. Enforcement of trusted nodes checking controlled by server's configuration variable `CheckTrustedNodes`. By default it is set to `1` and check of trusted nodes is enforced. To disable it, server's configuration variable `CheckTrustedNodes` must be set to `0`. The server restart is required to make this change effective.

Language syntax

Script entry point

NXSL handles script entry in 2 ways:

- Explicit main() function
- Implicit $main() function

If an explicitly defined main() exists, it will be called.

If an explicit main() doesn't exist, an implicit $main() function will be created by the script interpreter and the script will enter at the $main() function.

The $main() function is constructed from code that is not a part of any other functions.

Built-in Types

The following sections describe the standard types that are built into the interpreter.

NXSL is loose typed programming language. The system will automatically determine each variable type, assign a certain type to a variable and convert a variable type from one to another, if necessary. For example, a result for `3 + "4"` will be `7`, because the system will automatically convert `"4"` string into an integer. In case if the system is not able to automatically convert a line into an appropriate integer, the operation will result in a runtime error.

NXSL supports the following variable types:

- integer (32 bit),
- unsigned integer (32 bit),
- integer (64 bit), unsigned integer (64 bit),
- floating-point number,
- string,
- array,
- object.

In addition to that, NXSL also supports a special variable type – `NULL`. This value represents a variable with no value. `NULL` is the only possible value of type `NULL`. An attempt to perform any type of arithmetical or string operations with `NULL` variable will result in system runtime error.

It is possible to manually convert variable to a certain type, using a special function, named depending on the variable type. For example, `string(4)`. That way it is also possible to convert `NULL` type variables. Therefore, to avoid runtime errors while processing `NULL` type variables, it is advised to use manual conversion.

NXSL does not require setting variable type beforehand. The only exception to this is arrays. In case if an array is required, operator `array` defines its subsequent variables as arrays. Accessing variable which was not previously assigned will return `NULL` value.

Although NXSL has object type variables, it is not an object-oriented language. It is not possible to define classes or create objects at script level – only in extensions written in C++. Object type variables are used to return information about complex NetXMS objects, like nodes or events, in a convenient way. Please note that assigning object type variables actually holds reference to an object, so assigning object value to another variable does not duplicate actual object, but just copy reference to it.

To get a human-readable representation of a variable or expression type for debugging, use the `typeof()` function, and to get a class name for object type variables, use `classof()` function.

Truth Value Testing

Any object can be tested for truth value, for use in an if or while condition or as operand of the Boolean operations below. The following values are considered false:

False

NULL

zero of any numeric type, for example, 0, 0.0, 0j.

instances of user-defined classes, if the class defines a *bool*() or *len*() method, when that method returns the integer zero or bool value False. [1]

All other values are considered true — so objects of many types and arrays are always true.

Operations and built-in functions that have a Boolean result always return 0 or False for false and 1 or True for true, unless otherwise stated. (Important exception: the Boolean operations or and and always return one of their operands.)

Variables

Variables in NXSL behave the same way as variables in most popular programming languages (C, C++, etc.) do, but in NXSL you don’t have to declare variables before you use them.

Scope of a variable can be either global (visible in any function in the script) or local (visible only in the function within which it was defined). Any variable is by default limited to the local function scope. Variable can be declared global using `global` operator.

For example:

```
x = 1;
myFunction();

sub myFunction()
{
    println "x=" . x;
}
```

This script will cause run time error `Error 5 in line 6: Invalid operation with NULL value`, because variable `x` is local (in implicit main function) and is not visible in function `myFunction`. The following script will produce expected result (prints `x=1`):

```
global x = 1;
myFunction();

sub myFunction()
{
    println "x=" . x;
}
```

Function Declaration

A function is a named code block that is generally intended to process specified input values into an output value, although this is not always the case. For example, the `trace()` function takes variables and static text and prints the values into server log. Like many languages, NXSL provides for user-defined functions. These may be located anywhere in the main program or loaded in from other scripts via the use keywords.

To define a function, you can use the following form:

sub *NAME* **(** *ARGUMENTS* **) BLOCK**

where `NAME` is any valid identifier, `ARGUMENTS` is optional list of argument names, and `BLOCK` is code block.

To call a function you would use the following form:

NAME **(** *LIST* **)**

where `NAME` is identifier used in function definition, and `LIST` is an optional list of expressions passed as function arguments.

To give a quick example of a simple subroutine:

```
sub message()
{
    println "Hello!";
}
```

Function Arguments

The first argument you pass to the function is available within the function as `$1`, the second argument is `$2`, and so on. For example, this simple function adds two numbers and prints the result:

```
sub add()
{
    result = $1 + $2;
    println "The result was: " . result;
}
```

To call the subroutine and get a result:

```
add(1, 2);
```

If you want named arguments, list of aliases for `$1`, `$2`, etc. can be provided in function declaration inside the brackets:

```
sub add(numberA, numberB)
{
    result = numberA + numberB;
    println "The result was: " . result;
}
```

If parameter was not provided at function call, value of appropriate variable will be `NULL`.

Arguments also available as $ARGS array, that contains all arguments. First argument available as $ARGS[1];

Return Values from a Function

You can return a value from a function using the `return` keyword:

```
sub pct(value, total)
{
    return value / total * 100.0;
}
```

When called, return immediately terminates the current function and returns the value to the caller. If you don't specify a value in `return` statement or function ends implicitly by reaching end of function's block, then the return value is `NULL`.

Arrays

An array in NXSL is actually an ordered map. A map is a type that associates `values` to `keys`. This type is optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more. As array values can be other arrays.

A `key` must be a non-negative integer. When an array is created, its size is not specified and its map can have empty spots in it. For example, an array can have a element with a `0` key and an element with `4` key and no keys in-between. Attempting to access an array key which has not been defined is the same as accessing any other undefined variable: the result will be `NULL`.

Array elements can be accessed using [`index`] operator. For example, to access element with index `3` of array `a` you should use

```
a[3];
```

To get sub array form the array use `[a:b]` operator. This operator returns aub array of an array from the element with index `a` inclusive till the element with index `b` exclusive. If `a` is omitted then sub array will be taken form the start of the array and if `b` is omitted then sub array will be taken till the end of the array.

Example:

```
a = %(1, 2, 3, 4);
a2 = a[1:3]; // a2 will be %(2, 3)
```

Array Initialization

New array can be created in two ways. First is to use `'array' operator. This statement will create empty array and assign reference to it to variable `a`.

```
array a;
```

You can then assign values to the array like this.

Please note arrays in NXSL are sparse, so you can have elements with nothing in between.

```
array a;
a[1] = 1;
a[2] = 2;
a[260] = 260;
println(a[1]); // will print 1
```

Second way is to use %( ) construct to create array already populated with values.

This statement will create array with four elements at positions 0, 1, 2, and 3, and assign reference to this array to variable a.

```
// no need to use "array a;" here, since we are creating it directly
a = %(1, 2, 3, 4);

println(a[0]); // will actually print 1, since 1 is the 0th member
```

Array initialization can also be used directly in expressions, like this:

```
sub f()
{
    return %(2, "text", %(1, 2, 3));
}
```

In this example function `f` returns array of 3 elements - number, text, and another array of 3 numeric elements.

Operators

An operator is something that you feed with one or more values, which yields another value.

Arithmetic Operators

Example	Name	Result
`-a`	Negation	Opposite of `a`
`a + b`	Addition	Sum of `a` and `b`
`a - b`	Subtraction	Difference between `a` and `b`
`a * b`	Multiplication	Product of `a` and `b`
`a / b`	Division	Quotient of `a` and `b`
`a % b`	Modulus	Remainder of `a` divided by `b`

The division operator (`/`) returns a float value unless the two operands are integers (or strings that get converted to integers) and the numbers are evenly divisible, in which case an integer value will be returned.

Calling modulus on float operands will yield runtime error.

Assignment Operator

The assignment operator is `=`, which means that the left operand gets set to the value of the expression on the rights (that is, "gets set to").

Bitwise Operators

Example	Name	Result
`~ a`	Not	Bits that are set in `a` are unset, and vice versa.
`a & b`	And	Bits that are set in both operand are set.
`a | b`	Or	Bits that are set in either operand are set.
`a ^ b`	Xor	Bits that are set in only one operand are set.
`a << b`	Shift left	Shift the bits of `a` for `b` steps to the left (each step equals "multiply by two").
`a >> b`	Shift right	Shift the bits of `a` for `b` steps to the right (each step equals "divide by two").

Comparison Operators

Comparison operators allow you to compare two values.

Example	Name	Result
`a == b`	Equal	`TRUE` if `a` is equal to `b`.
`a != b`	Not equal	`TRUE` if `a` is not equal to `b`.
`a < b`	Less than	`TRUE` if `a` is strictly less than `b`.
`a > b`	Greater than	`TRUE` if `a` is strictly greater than `b`.
`a <= b`	Less than or equal to	`TRUE` if `a` is less than or equal to `b`.
`a >= b`	Greater than or equal to	`TRUE` if `a` is greater than or equal to `b`.

Example	Name	Result
`a ~= b`	Match	Array with full match of b in the position 0 and other matches starting from 1st position if `a` is matched to regular expression `b` or NULL if not matched. As a side effect, assigns values to special variables `$1`, `$2`, `$3`, etc. See see Regular expressions for details. System message: Unknown interpreted text role "ref".
`a match b`	Match	Array with full match of b in the position 0 and other matches starting from 1st position if `a` is matched to regular expression `b` or NULL if not matched. As a side effect, assigns values to special variables `$1`, `$2`, `$3`, etc. See see Regular expressions for details. System message: Unknown interpreted text role "ref".
`a imatch b`	Match (case insensitive)	Array with full match of b in the position 0 and other matches starting from 1st position if `a` is matched to regular expression `b` (case insensitive) or NULL if not matched. As a side effect, assigns values to special variables `$1`, `$2`, `$3`, etc. See see Regular expressions for details. System message: Unknown interpreted text role "ref".

Incrementing/Decrementing Operators

NXSL supports C-style pre- and post-increment and decrement operators.

Example	Name	Result
`++a`	Pre-increment	Increments `a` by one, then returns `a`.
`a++`	Post-increment	Returns `a`, then increments `a` by one.
`--a`	Pre-decrement	Decrements `a` by one, then returns `a`.
`a--`	Post-decrement	Returns `a`, then decrements `a` by one.

Logical Operators

Example	Name	Result
`! a`	Not	`TRUE` if `a` is not `TRUE`.
`a && b`	And	`TRUE` if both `a` and `b` is `TRUE`.
`a || b`	Or	`TRUE` if either `a` or `b` is `TRUE`.

String Operators

Example	Name	Result
`.`	Concatenation operator	Returns the concatenation of its right and left arguments.
`.=`	Concatenating assignment operator	Appends the argument on the right side to the argument on the left side.
`[a:b]`	Substring operator	Returns substring of a string from the character with index `a` inclusive till the character with index `b` exclusive. Example: "1234"[1:3] will be "23". If `a` is omitted then substring will be taken form the start of the string and if `b` is omitted then substring will be taken till the end of the string.

Control structures

Any NXSL script is built out of a series of statements. A statement can be an assignment, a function call, a loop, a conditional statement or even a statement that does nothing (an empty statement). Statements usually end with a semicolon. In addition, statements can be grouped into a statement-group by encapsulating a group of statements with curly braces. A statement-group is a statement by itself as well. The various statement types are supported:

- if
- else
- while
- do-while
- for
- break
- continue
- switch
- return
- exit

if

The `if` construct is one of the most important features of many languages. It allows for conditional execution of code fragments. NXSL features an `if` structure that is similar to that of C:

```
if (expr)
    statement
```

else

Often you'd want to execute a statement if a certain condition is met, and a different statement if the condition is not met. This is what `else` is for. `else` extends an `if` statement to execute a statement in case the expression in the `if` statement evaluates to `FALSE`. The `else` statement is only executed if the `if` expression evaluated to `FALSE`.

while

`while` loops are the simplest type of loop in NXSL. They behave just like their C counterparts. The basic form of a `while` statement is:

```
while (expr)
    statement
```

The meaning of a `while` statement is simple. It tells NXSL to execute the nested statement(s) repeatedly, as long as the `while` expression evaluates to `TRUE`. The value of the expression is checked each time at the beginning of the loop, so even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration.

do-while

`do-while` loops are very similar to `while` loops, except the truth expression is checked at the end of each iteration instead of in the beginning. The main difference from regular `while` loops is that the first iteration of a `do-while` loop is guaranteed to run (the truth expression is only checked at the end of the iteration), whereas it may not necessarily run with a regular `while` loop (the truth expression is checked at the beginning of each iteration, if it evaluates to `FALSE` right from the beginning, the loop execution would end immediately).

for

`for` loops are the most complex loops in NXSL. They behave in two different ways: like their C counterparts or in Java way. The syntax of a `for` loop is:

```
for (expr1; expr2; expr3)
    statement

for (varName : array)
    statement
```

The first expression (`expr1`) is evaluated (executed) once unconditionally at the beginning of the loop.

In the beginning of each iteration, `expr2` is evaluated. If it evaluates to `TRUE`, the loop continues and the nested statement(s) are executed. If it evaluates to `FALSE`, the execution of the loop ends.

At the end of each iteration, `expr3` is evaluated (executed).

In the second example for cycle will call `statement` for each element in array. Element will be available as `varName`.

break

`break` ends execution of the current `for`, `while`, `do-while` or `switch` structure.

continue

`continue` is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration.

switch

The `switch` statement is similar to a series of `if` statements on the same expression. In many occasions, you may want to compare the same variable (or expression) with many different values, and execute a different piece of code depending on which value it equals to. This is exactly what the `switch` statement is for.

Example:

```
switch (input)
{
  case "1":
    trace(0,"Input is 1");
    break;
  case "2":
    trace(0,"Input is 2");
    break;
  default:
    trace(0, "Input is unknown");
}
```

return

If called from within a function, the `return` statement immediately ends execution of the current function, and returns its argument as the value of the function call. Calling `return` from `main()` function (either explicitly or implicitly defined) is equivalent of calling `exit`.

exit

The `exit` statement immediately ends execution of the entire script, and returns its argument as script execution result.

Expressions

The simplest yet most accurate way to define an expression is "anything that has a value".

The most basic forms of expressions are constants and variables. When you type `a = 5`, you’re assigning `5` into `a`. `5`, obviously, has the value 5, or in other words `5` is an expression with the value of 5 (in this case, `5` is an integer constant).

Slightly more complex examples for expressions are functions. Functions are expressions with the value of their return value.

NXSL supports the following value types: integer values, floating point values (float), string values and arrays. Each of these value types can be assigned into variables or returned from functions.

Another good example of expression orientation is pre- and post-increment and decrement. You be familiar with the notation of `variable++` and `variable--`. These are increment and decrement operators. In NXSL, like in C, there are two types of increment - pre-increment and post-increment. Both pre-increment and post-increment essentially increment the variable, and the effect on the variable is identical. The difference is with the value of the increment expression. Pre-increment, which is written `++variable`, evaluates to the incremented value. Post-increment, which is written `variable++` evaluates to the original value of variable, before it was incremented.

A very common type of expressions are comparison expressions. These expressions evaluate to either `FALSE` or `TRUE`. NXSL supports `>` (bigger than), `>=` (bigger than or equal to), `=` (equal), `!=` (not equal), `<` (less than) and `<=` (less than or equal to). These expressions are most commonly used inside conditional execution, such as `if` statements.

The last example of expressions is combined operator-assignment expressions. You already know that if you want to increment `a` by 1, you can simply write `a++` or `++a`. But what if you want to add more than one to it, for instance 3? In NXSL, adding 3 to the current value of `a` can be written `a += 3`. This means exactly "take the value of `a`, add 3 to it, and assign it back into `a` ". In addition to being shorter and clearer, this also results in faster execution. The value of `a += 3`, like the value of a regular assignment, is the assigned value. Notice that it is NOT 3, but the combined value of `a` plus 3 (this is the value that's assigned into `a`). Any two-place operator can be used in this operator-assignment mode.

Short-circuit evaluation

Short-circuit evaluation denotes the semantics of some Boolean operators in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression: when the first argument of the AND function evaluates to false, the overall value must be false; and when the first argument of the OR function evaluates to true, the overall value must be true. NXSL uses short-circuit evaluation for `&&` and `||` boolean operators. This feature permits two useful programming constructs. Firstly, if the first sub-expression checks whether an expensive computation is needed and the check evaluates to false, one can eliminate expensive computation in the second argument. Secondly, it permits a construct where the first expression guarantees a condition without which the second expression may cause a run-time error. Both are illustrated in the following example:

```
if ((x != null) && ((trim(x) == "abc") || (long_running_test(x)))
   do_something();
```

Without short-circuit evaluation, `trim(x)` would cause run-time error if `x` is `NULL`. Also, long running function will only be called if condition (`trim(x) == "abc"`) will be false.

Regular expressions

Since version 3.0, regular expression engine is changed to PCRE (Perl compatible). Syntax can be checked with `pcregrep`, perl itself or on regex101.com (select PCRE flavour).

Comments

Tutorial

Syntactically, NXSL looks similar to Perl or C. Here's simple NXSL program:

```
/* sample program */
sub main()
{
    println "Hello!";
    return 0;
}
```

This program will print word `Hello` on screen.

Also, keep in mind that you are free to choose your own formatting style. E.g. the above could have been written as:

```
/* sample program */ sub main(){println "Hello!";return 0;}
```

Now we'll analyze this program:

```
/* sample program */
```

Everything inside `/* */` is considered a comment and will be ignored by interpreter. You can enclose comments, like below:

```
/* comment /* another comment */ still comment */
```

You can also use single line comments:

```
x = 1; // everything between two slashes and end of line is a comment
```

Now onto next line:

```
sub main()
{
}
```

This is a function definition. A function is a part of a program that can be called by other parts of the program. A function definition always has the following form:

sub *name* **(***parameters***)** {

```
\\the function code goes here
```

}

The function can return a value to the caller and accept zero or more parameters.

The function name follows the rules for all names (formally: identifiers): it must consist entirely of letters (uppercase and lowercase are different!), digits, underscores (`_`) and dollar signs (`$`), but may not begin with a digit. Please note that most special identifiers starts with dollar sign (`$`), so it is recommended not to start your identifiers with it.

First line in function code looks like

```
println "Hello!";
```

In this line, `println` is an embedded operator which prints given string to standard output with carriage return, and `"Hello!"` is a string we want to print. Please note semicolon at the end of line – it's a separator between operators. Each operator should end with semicolon.

The next, and final, line of our small program is:

```
return 0;
```

`return` is another built-in operator which exits the function and sets it's return value.

Function Reference

String functions

ArrayToString()

```
ArrayToString() => void
```

…

Parameters

Example

```
```

chr()

```
chr() => void
```

…

Parameters

Example

```
```

d2x()

```
d2x(number, padding=0) => String
```

Convert decimal `devValue` to hex string with optional left-padding with zeroes.

Parameters

number	Input value.
padding	Optional argument specifying target string length.

Return

Hex string.

Example

```
>>> d2x(1234)
4D2
>>> d2x(1234, 8)
000004D2
```

format()

```
format() => void
```

...

Parameters

Example

```
```

index()

```
index() => void
```

...

Parameters

Example

```
```

inList()

```
inList(string, separator, token) => Boolean
```

Split input `string` by `separator` into elements and compare each element with `token`.

Parameters

string	Input string.
separator	Elements separator.
token	Pattern to compare with.

Return

`True` if token is found in input string.

Example

```
>>> inList("1,2,3", ",", "1")
true
>>> inList("ab|cd|ef", "|", "test")
false
```

left()

```
left() => void
```

…

Parameters

Example

```
```

length()

```
length() => void
```

…

Parameters

Example

```
```

lower()

```
lower() => void
```

...

Parameters

Example

```
```

ltrim()

```
ltrim() => void
```

...

Parameters

Example

```
```

ord()

```
ord() => void
```

...

Parameters

Example

```
```

right()

```
right() => void
```

...

Parameters

Example

```
```

rindex()

```
rindex() => void
```

...

Parameters

Example

```
```

rtrim()

```
rtrim() => void
```

...

Parameters

Example

```
```

SplitString()

```
SplitString() => void
```

...

Parameters

Example

```
```

substr()

```
substr() => void
```

...

Parameters

Example

```
```

trim()

```
trim() => void
```

...

Parameters

Example

```
```

upper()

```
upper() => void
```

...

Parameters

Example

```
```

x2d()

```
x2d(hexValue) => Integer
```

Convert hexadecimal string to decimal value.

Parameters

hexValue	Input value.

Return

Converted value.

Example

```
>>> x2d("4D2")
1234
```

Math functions

abs()

```
abs(number) => Number
```

Returns the absolute value of the number.

Parameters

number	Input value.

Return

Absolue value of the input.

Example

```
>>> abs(12.3)
12.3
>>> abs(-0.307)
0.307
```

acos()

```
acos() => void
```

...

Parameters

Example

```
```

asin()

```
asin() => void
```

…

Parameters

Example

```
```

ceil()

```
ceil(input) => Integer
```

Round up value.

Parameters

input	Input value.

Return

Value round up to nearest integer.

Example

```
>>> ceil(2.3)
3.0
>>> ceil(3.8)
4.0
>>> ceil(-2.3)
-2.0
>>> ceil(-3.8)
-3.0
```

cos()

```
cos() => void
```

…

Parameters

Example

```
```

cosh()

```
cosh() => void
```

...

Parameters

Example

```
```

exp()

```
exp(input) => Float
```

Computes `e**x`, the base-e exponential.

Parameters

input	Input number.

Example

```
>>> exp(2)
7.3890561
```

floor()

```
floor(input) => Integer
```

Round down value.

Parameters

input	Input value.

Return

Value round down to nearest integer.

Example

```
>>> floor(2.3)
2
>>> floor(3.8)
3
>>> floor(-2.3)
-3
>>> floor(-3.8)
-4
```

log()

```
log() => void
```

...

Parameters

Example

```
```

log10()

```
log10() => void
```

...

Parameters

Example

```
```

max()

```
max() => void
```

...

Parameters

Example

```
```

min()

```
min() => void
```

...

Parameters

Example

```
```

pow()

```
pow() => void
```

...

Parameters

Example

```
```

round()

```
round() => void
```

...

Parameters

Example

```
```

sin()

```
sin() => void
```

...

Parameters

Example

```
```

sinh()

```
sinh() => void
```

...

Parameters

Example

```
```

tan()

```
tan() => void
```

...

Parameters

Example

```
```

tanh()

```
tanh() => void
```

...

Parameters

Example

```
```

Time related functions

gmtime()

```
gmtime() => void
```

…

Parameters

Example

```
```

localtime()

```
localtime() => void
```

…

Parameters

Example

```
```

mktime()

```
mktime() => void
```

…

Parameters

Example

```
```

strftime()

```
strftime() => void
```

…

Parameters

Example

```
```

time()

```
time() => void
```

…

Parameters

Example

```
```

Object functions

BindObject()

Deprecated since 3.0, use NetObj::bind() or NetObj::bindTo() instead.

This function is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

```
BindObject(parent, child) => void
```

Bind all NetXMS objects that can be bound from console (nodes, subnets, clusters, and another containers) to container objects.

Parameters

parent	Parent object (NetObj referring to container object or infrastructure service root).
child	The NetXMS object to be linked to given parent object (Node or NetObj referring to subnet, container, or cluster).

Return

None.

Example

```
BindObject(FindObject(2), $node);    // Link current node directly to "Infrastructure
Services"
BindObject(FindObject("Services"), FindObject("Service_1"));    // Link object named
"Service_1" to container "Services"
```

CreateContainer()

This function is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

```
CreateContainer(parent, name) => Container
```

Create new container under `parent` object with desired `name`.

Parameters

parent	Parent object (NetObj referring to container object or infrastructure service root).
name	Name of the container to create

Return

Instance of newly created Container object or `null` if failed.

Example

```
```

CreateNode()

```
CreateNode() => void
```

AF_ENABLE_NXSL_CONTAINER_FUNCS

…

Parameters

Example

```
```

DeleteCustomAttribute()

Deprecated since 3.0, use NetObj::deleteCustomAttribute() instead.

```
DeleteCustomAttribute(object, name) => void
```

Delete custom attribute `name` from `object`.

Parameters

object	Target object.
name	Name of the custom attribute.

Example

```
>>> DeleteCustomAttribute($node, "test")
>>> test@$node
null
```

DeleteObject()

Deprecated since 3.0, use NetObj::delete() instead.

This function is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

```
DeleteObject(object) => void
```

Delete object of class NetObj, Interface, or Node from the system.

Parameters

object	NetXMS object to be deleted. Can be instance of NetObj or any inherited (e.g. Node). Reference to the object can be obtained using FindObject() function.

Return

None.

Example

```
DeleteObject(FindObject("Service_1")); //delete object named Service_1
```

EnterMaintenance()

Deprecated since 3.0, use NetObj::enterMaintenance() instead.

```
EnterMaintenance(object) => void
```

Make an object enter Maintenance mode.

Parameters

object	NetObj that will be entered in maintenance mode

Return

None.

Example

```
EnterMaintenance($node);    // Enter current node in maintenance mode
EnterMaintenance(FindObject("Services"));    // Enter container "Services" in
maintenance mode
```

FindNodeObject()

```
FindNodeObject(currentNode, key) => Node
```

Look up Node object by either name or object id, will return `null` if object not found or not accessible. This function search for nodes only.

If trusted node validation is enforced, `currentNode` should point to execution context object (instance of NetObj, `$node` in most cases). If trusted nodes are disabled (default server configuration), `currentNode` can be set to `null`.

Parameters

currentNode	Lookup context or `null` if trusted nodes validation is disabled.
key	Object name of id.

Return

Instance of Node object or `null` if not found or not accessible.

Example

```
>>> FindNodeObject($node, "server.netxms.org")
object
>>> FindNodeObject(null, 12)
object
>>> FindNodeObject($node, "bad_node_name")
NULL
```

FindObject()

```
FindObject(currentNode, key) => NetObj
```

Look up any object inherited from NetObj (Interface, Cluster, etc.) by either name or object id.

If trusted node validation is enforced, `currentNode` should point to execution context object (instance of NetObj, `$node` in most cases). If trusted nodes are disabled (default server configuration), `currentNode` can be set to `null`.

Parameters

currentNode	Lookup context or `null` if trusted nodes validation is disabled.
key	Object name of id.

Return

Instance of object inherited from NetObj or `null` if not found or not accessible. Type of the object can be verified using function classof().

Example

```
```

GetAllNodes()

```
GetAllNodes() => array
```

Get list of all Node objects in the system as array.

Return

Array of node objects.

Example

```
>>> for (n : GetAllNodes()) {
>>>   println(n->id . " - " . n->name);
>>> }

6766 - demo-netxms
6901 - Control Unit 1
6902 - Control Unit 2
```

GetCustomAttribute()

Deprecated since 3.0, use NetObj::getCustomAttribute() instead.

```
GetCustomAttribute(object, name) => String
```

Lookup custom attribute of the object.

Alternatively, attributes can be accessed as instance attribues (with →, attribute should exist) or by using `attribute@object` notion (which will return `null` instead of runtime error if attribue is missing).

Parameters

object	Object to query.
name	Name of the custom attribute.

Return

String value of the custom attribute of `null` if not found.

Example

```
>>> GetCustomAttribute($node, "test")
testvalue
>>> $node->test
testvalue
>>> test@$node
testvalue
```

GetInterfaceName()

```
GetInterfaceName() => void
```

…

Parameters

Example

```
```

GetInterfaceObject()

```
GetInterfaceObject() => void
```

…

Parameters

Example

```
```

GetNodeInterfaces()

> ❗ **This function is deprecated starting from version 3.0.** Please use interfaces attribute in Node.

```
GetNodeInterfaces(node) => void
```

Get all interfaces for given node.

Parameters

node	Object of class Node.

Return

Array of objects of class Interface, with first object placed at index 0.

Example

```
// Log names and ids of all interface objects for given node
interfaces = GetNodeInterfaces($node);
for(i : interfaces)
{
    trace(1, "Interface: name='" . i->name . "' id=" . i->id);
}
```

GetNodeParents()

```
GetNodeParents() => void
```

…

Parameters

Example

```
```

GetNodeTemplates()

> ❗ **This function is deprecated starting from version 3.0.** Please use templates attribute of [class-DataCollectionTarget].

```
GetNodeTemplates()
```

Get template objects applied on given node.

Parameters

node	Node object.

Return

Array of objects, with first object placed at index 0. Return value also affected by trusted nodes settings.

GetObjectChildren()

This function is deprecated starting from version 3.0. Please use children attribute in NetObj.

```
GetObjectChildren(object) => Array
```

Return array of child objects for the object.

Parameters

object	Target object.

Return

Array of NetObj instances.

Example

```
// Log names and ids of all accessible child objects for current node
children = GetObjectChildren($node);
for(p : children)
{
    trace(1, "Child object: name='" . p->name . "' id=" . p->id);
}
```

GetObjectParents()

This function is deprecated starting from version 3.0. Please use parents attribute in NetObj.

```
GetObjectParents(object) => Array
```

Return array of object parents.

Parameters

object	Target object.

Return

Array of NetObj instances.

Example

```
// Log names and ids of all accessible parents for current node
parents = GetObjectParents($node);
for(p : parents)
{
    trace(1, "Parent object: name='" . p->name . "' id=" . p->id);
}
```

LeaveMaintenance()

Deprecated since 3.0, use NetObj::leaveMaintenance() instead.

```
LeaveMaintenance(object) => void
```

Make an object leave Maintenance mode.

Parameters

object	NetObj that will leave maintenance mode

Return

None.

Example

```
LeaveMaintenance($node);    // Make current node leave maintenance mode
LeaveMaintenance(FindObject("Services"));    // Make container "Services" leave
maintenance mode
```

ManageObject()

This function is deprecated starting from version 3.0. Please use manage functions in NetObj.

```
ManageObject(object) => void
```

Set object into managed state. Has no effect if object is already in managed state.

Parameters

object	NetXMS object to be modified. Can be NXSL class NetObj or any inherited for it. Reference to object can be obtained using FindObject() function.

Example

```
ManageObject(FindObject(125));    // Set object with id 125 to managed state
```

RenameObject()

This function is deprecated starting from version 3.0. Please use unbind and unbindFrom functions in NetObj.

```
RenameObject(object, name) => void
```

Rename object.

Parameters

object	NetXMS object to be renamed. Can be NXSL class NetObj or any inherited for it. Reference to object can be obtained using FindObject function.
name	New name for object.

Return

None.

Example

```
RenameObject(FindObject(2), "My Services");    // Rename "Infrastructure Services"
object
```

SetCustomAttribute()

This function is deprecated starting from version 3.0. Please use setCustomAttribute functions in NetObj.

```
SetCustomAttribute(object, name, value) => void
```

Set custom attribute `name` to `value` on `object`.

Parameters

object	Target object.
name	Custom attribute name.
value	Custom attribute value.

Example

```
>>> SetCustomAttribute($node, "test", "test value")
>>> test@$node
test value
```

SetInterfaceExpectedState()

This function is deprecated starting from version 3.0. Please use setExpectedState functions in Interface.

```
SetInterfaceExpectedState() => void
```

Set expected state for given interface.

Parameters

interface	Interface object. Can be obtained using GetNodeInterfaces or GetInterfaceObject.
state	New expected state for interface. Can be specified as integer code or state name. Interface expected states

Return

None.

Example

```
// Set expected state to "ignore" for all interfaces of given node
interfaces = GetNodeInterfaces($node);
foreach(i : interfaces)
{
    SetInterfaceExpectedState(i, "IGNORE");
}
```

UnbindObject()

Deprecated since 3.0, use NetObj::unbind() and NetObj::unbindFrom() instead.

This function is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

```
UnbindObject(parent, child) => void
```

Remove (unbind) object from a container.

Parameters

parent	Parent object (NetObj referring to container object or infrastructure service root).
child	The NetXMS object to be unlinked from given parent object (Node or NetObj referring to node, subnet, container, or cluster).

Return

None.

Example

```
UnbindObject(FindObject("Services"), FindObject("Service_1"));    // Unlink object
named "Service_1" from container "Services"
```

UnmanageObject()

This function is deprecated starting from version 3.0. Please use unmanage functions in NetObj.

```
UnmanageObject(object) => void
```

Set object into unmanaged state. Has no effect if object is already in unmanaged state.

Parameters

object	NetXMS object to be modified. Can be NXSL class NetObj, Node, or Interface. Reference to object can be obtained using FindObject function.

Return

None.

Example

```
UnmanageObject(FindObject(2));    // Set "Infrastructure Services" object to unmanaged
state
```

Data Collection

CreateDCI()

```
CreateDCI(node, origin, name, description, dataType, pollingInterval, retentionTime)
=> DCI
```

Create new data collection item on node, return DCI object instance of `null` if failed.

Parameters

node	Node object instance (e.g. `$node`), where DCI should be created.
origin	data origin, supported values: • "internal" • "agent" • "snmp" • "cpsnmp" • "push" • "winperf" • "smclp" • "script" • "ssh" • "mqtt" • "driver"
name	name of the metric (e.g. "Agent.Version")
description	human readable description.

dataType	type of the collected data, supported values: • "int32" • "uint32" • "int64" • "uint64" • "string" • "float".
pollingInterval	polling interval in seconds or `0` for server-default value.
retentionTime	retention time in days or `0` for server-default value.

Return

Instance of newly created DCI of `null` if failed.

Example

```
>>> d = CreateDCI($node, "agent", "Agent.Version", "Agent Version", "string", 0, 0);
>>> println(d->id);
145
```

FindAllDCIs()

```
FindAllDCIs(node, nameFilter, descriptionFilter) => Array
```

Find all DCI on the `node` matching `nameFilter` **and** `descriptionFilter`. Filter can contain glob symbols "?" and "*". If filter is `null`, it's ignored.

Parameters

node	Node object instance (e.g. `$node`)
nameFilter	GLOB for mastching DCI name or `null` if name should be ignored.
descriptionFilter	GLOB for mastching DCI description or `null` if description should be ignored.

Return

Array of DCI.

Example

```
>>> list = FindAllDCIs($node, "Server*", "*MAIN*");
>>> foreach (row : list) {
>>>     println(row->id . ": " . row->description . " (" . row->name . ")");
>>> }
91: Server thread pool MAIN: usage (Server.ThreadPool.Usage(MAIN))
92: Server thread pool MAIN: normalized load average (1 minute) (Server.ThreadPool
.LoadAverage(MAIN,1))
93: Server thread pool MAIN: current load (Server.ThreadPool.Load(MAIN))

>>> list = FindAllDCIs($node, "Server*");
>>> foreach (row : list) {
>>>     println(row->id . ": " . row->description . " (" . row->name . ")");
>>> }
100: NetXMS server: database writer's request queue (other queries) (Server
.AverageDBWriterQueueSize.Other)
101: NetXMS server: database writer's request queue (Server.AverageDBWriterQueueSize)
103: NetXMS server: data collector's request queue (Server
.AverageDataCollectorQueueSize)
...

>>> list = FindAllDCIs($node, null, "*load average*");
>>> foreach (row : list) {
>>>     println(row->id . ": " . row->description . " (" . row->name . ")");
>>> }
119: CPU: load average (15 minutes) (System.CPU.LoadAvg15)
123: CPU: load average (5 minutes) (System.CPU.LoadAvg5)
83: Server thread pool AGENT: normalized load average (1 minute) (Server.ThreadPool
.LoadAverage(AGENT,1))
...
```

FindDCIByDescription()

```
FindDCIByDescription(node, description) => Integer
```

Find ID of the DCI on node by description (exact match). FindAllDCIs() can be used for pattern search.

Parameters

node	Node object instance (e.g. `$node`)
description	Description of the DCI

Return

Integer ID of the DCI or `null` if not found.

Example

```
>>> d = FindDCIByDescription($node, "Agent Version");
>>> print(d);
144
```

FindDCIByName()

```
FindDCIByName(node, dciName) => Integer
```

Find ID of the DCI on node by name (exact match). FindAllDCIs() can be used for pattern search.

Parameters

node	Node object instance (e.g. `$node`)
dciName	Name of the DCI

Return

Integer ID of the DCI or `null` if not found.

Example

```
>>> d = FindDCIByName($node, "Agent.Version");
>>> print(d);
144
```

GetAvgDCIValue()

```
GetAvgDCIValue(object, dciId, periodStart, periodEnd) => Number
```

Get the average value of the DCI for the given period. The DCI value type must be numeric.

Parameters

object	Instance of Node, Cluster, or MobileDevice object (e.g. `$node`).
dciId	ID of the DCI to retrive.
periodStart	Unix timestamp of the period start.
periodEnd	Unix timestamp of the period end.

Return

Average value or `null` on failure.

Example

```
>>> obj = FindObject("Server1");
>>> dciID = FindDCIByName(obj, "CPU.Usage")
>>> val = GetAvgDCIValue(obj, dciId, 0, time()); // time range from January 1, 1970
untill now
>>> println("Average CPU Usage: ". val . "%");
Average CPU Usage: 17%
```

GetDCIObject()

```
GetDCIObject() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetDCIRawValue()

```
GetDCIRawValue() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetDCIValue()

```
GetDCIValue(node, dciId) => String or Table
```

Get last collected value of the DCI. Return `null` on error, Table instance for table DCI or String otherwise.

Parameters

node	Node object instance (e.g. `$node`)

Return

Table for table DCIs, String, or `null` if failed or no data is available.

Example

```
>>> GetDCIValue($node, FindDCIByName($node, "Status"))
0
>>> GetDCIValue($node, FindDCIByName($node, "Non-Existing"))
null
```

GetDCIValueByDescription()

```
GetDCIValueByDescription() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetDCIValueByName()

```
GetDCIValueByName() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetDCIValues()

```
GetDCIValues() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetMaxDCIValue()

```
GetMaxDCIValue() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetMinDCIValue()

```
GetMinDCIValue() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

GetSumDCIValue()

```
GetSumDCIValue() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

PushDCIData()

```
PushDCIData() => Boolean
```

…

Parameters

node	Node object instance (e.g.)

Example

```
```

Agent related functions

AgentExecuteAction()

```
AgentExecuteAction(node, actionName, …) => Boolean
```

Execute agent action on given node. Optional arguments starting from 3rd are passed as action arguments to the agent.

Parameters

node	Node object instance (e.g. `$node`)
actionName	Name of the action to be executed
…	Optional arguments for action

Return

Boolean indicator of success

Example

```
>>> AgentExecuteAction($node, "System.Restart");
true

>>> AgentExecuteAction($node, "Custom.RestartService", "jetty9");
true

>>> AgentExecuteAction($node, "nonexisting action");
false
```

AgentExecuteActionWithOutput()

```
AgentExecuteActionWithOutput(node, actionName, …) => String
```

Execute agent action on given node and collect standard output of the application defined by action. Optional arguments starting from 3rd are passed as action arguments to the agent.

Parameters

node	Node object instance (e.g. `$node`)
actionName	Name of the action to be executed
…	Optional arguments for action

Return

Output of the action or `null` if execution failed.

Example

```
>>> AgentExecuteActionWithOutput($node, "Custom.Ping", "10.10.8.16");
PING 10.10.8.16 (10.10.8.16): 56 data bytes
64 bytes from 10.10.8.16: icmp_seq=0 ttl=64 time=0.084 ms
64 bytes from 10.10.8.16: icmp_seq=1 ttl=64 time=0.120 ms
64 bytes from 10.10.8.16: icmp_seq=2 ttl=64 time=0.121 ms
```

AgentReadList()

This function is deprecated starting from version 3.0. Please use readAgentList function in Node.

```
AgentReadList(node, name) => Array
```

Request list metric directly from agent on given node.

Parameters

node	Node object instance (e.g. `$node`)
name	List name

Return

Array of strings or `null` if failed.

Example

```
>>> supportedLists = AgentReadList($node, "Agent.SupportedLists");
>>> foreach (l : supportedLists) { println(l); }
Agent.ActionList
Agent.SubAgentList
Agent.SupportedLists
Agent.SupportedParameters
Agent.SupportedPushParameters
...
```

AgentReadParameter()

❗ **This function is deprecated starting from version 3.0.** Please use readAgentParameter function in Node.

```
AgentReadParameter(node, name) => String
```

Request metric directly from agent on given node.

Parameters

node	Node object instance (e.g. `$node`)
name	Metric name

Return

String value or `null` if failed.

Example

```
>>> v = AgentReadParameter($node, "Agent.Version")
>>> println(v)
2.2.13
```

AgentReadTable()

❗ **This function is deprecated starting from version 3.0.** Please use readDriverParameter function in Node.

```
AgentReadTable(node, name) => Table
```

Request table metric directly from agent on given node.

Parameters

node	Node object instance (e.g. `$node`)
name	List name

Return

Instance of Table or `null` if failed.

Example

```
>>> t = AgentReadTable($node, "Agent.SubAgents");
>>> for (c : t->columns) {
>>>   print(c->name . " | ");
>>> }
>>> println("");
>>> for (row : t->rows) {
>>>   for(cell : row->values) {
>>>      print(cell . " | ");
>>>   }
>>>   println("");
>>> }
NAME | VERSION | FILE |
Darwin | 2.2.13 | darwin.nsm |
FILEMGR | 2.2.13-3-g4c02b65c50 | filemgr.nsm |
PING | 2.2.13-3-g4c02b65c50 | ping.nsm |
```

Alarm functions

FindAlarmById()

```
FindAlarmById() => void
```

…

Parameters

Example

```

```

FindAlarmByKey()

```
FindAlarmByKey() => void
```

…

Parameters

Example

```

```

FindAlarmByKeyRegex()

```
FindAlarmByKeyRegex() => void
```

...

Parameters

Example

```

```

Events

GetEventParameter()

```
GetEventParameter() => void
```

...

Parameters

Example

```

```

SetEventParameter()

```
SetEventParameter() => void
```

...

Parameters

Example

```

```

LoadEvent()

```
LoadEvent(eventId) => <<class-event>>
```

Will load event form the database and return Event object or NULL if event not found.

Parameters

eventId	Id of the event to be loaded form database

Example

```
>>>event = LoadEvent(315);
>>>event->id;
315
```

PostEvent()

```
PostEvent() => void
```

...

Parameters

Example

```
```

Miscelanious functions

_exit()

```
_exit(exitCode=0) => void
```

Stop script execution and return `exitCode`.

Parameters

exitCode	Optional exit code for the script. Defaults to `0`.

AddrInRange()

```
AddrInRange() => void
```

...

Parameters

Example

```
```

AddrInSubnet()

```
AddrInSubnet() => void
```

...

Parameters

Example

```
```

assert()

```
assert() => void
```

...

Parameters

Example

```
```

classof()

```
classof(instance) => String
```

Return class name of the `instance`.

Parameters

instance	Instance of any class.

Return

Class name.

Example

```
>>> classof($node)
Node
```

CountryAlphaCode()

```
CountryAlphaCode(code) => String
```

Lookup country alpha code by numeric or alpha3 code.

Parameters

code	Numeric (3 digits) or 3-letter country code.

Return

Two letter country code or `null` if country not found.

Example

```
>>> CountryAlphaCode('020')
AN
>>> CountryAlphaCode('AND')
AN
>>> CountryAlphaCode('124')
CA
```

CountryName()

```
CountryName() => void
```

…

Parameters

Example

```
```

CountScheduledTasksByKey()

```
CountScheduledTasksByKey() => void
```

…

Parameters

Example

```
```

CreateUserAgentNotification()

Creates user agent notification

```
CreateUserAgentNotification(object, message, startTime, endTime) => Number
```

…

Parameters

object	Node or root object to send notification
message	Message to be sent to clients
startTime	Start time of notification delivery
endTime	End time of notification delivery

Return

New user agent notification id

Example

```
>>> AgentExecuteAction($node, "One time notification text", 0, 0);
14

>>> AgentExecuteAction($node, "Interval user agent notification text", time(),
time()+86400);
15
```

CurrencyAlphaCode()

```
CurrencyAlphaCode() => void
```

…

Parameters

Example

```
```

CurrencyExponent()

```
CurrencyExponent() => void
```

…

Parameters

Example

```
```

CurrencyName()

```
CurrencyName() => void
```

…

Parameters

Example

```
```

DriverReadParameter()

ℹ Deprecated since 3.0, use Node::readDriverParameter() instead.

```
DriverReadParameter(name) => String
```

Request driver-specific metric directly from network device driver (e.g. Rital). Works similarly to AgentReadParameter(), but query device driver insterad.

Parameters

name	Name of the metric to query

Return

String value of the metric or `null` if request failed / metric not supported by driver

Example

```
```

EventCodeFromName()

```
EventCodeFromName() => void
```

…

Parameters

Example

```
```

EventNameFromCode()

```
EventNameFromCode() => void
```

…

Parameters

Example

```
```

GetConfigurationVariable()

```
GetConfigurationVariable(key, defaultValue=null) => void
```

Read server configuration parameter by `key`.

Parameters

key	Configuration parameter name to lookup.
defaultValue	Optional argument with default value if key is not found.

Return

Value of the server configuration parameter. If key is not found, `null` is returned or `defaultValue` is specified.

Example

```
>>> GetConfigurationVariable("NumberOfStatusPollers")
10
>>> GetConfigurationVariable("BadVariable")
NULL
>>> GetConfigurationVariable("BadVariable", 22)
22
```

gethostbyaddr()

```
gethostbyaddr() => void
```

…

Parameters

Example

```
```

gethostbyname()

```
gethostbyname() => void
```

…

Parameters

Example

```
```

GetSyslogRuleCheckCount()

```
GetSyslogRuleCheckCount() => void
```

…

Parameters

Example

```
```

GetSyslogRuleMatchCount()

```
GetSyslogRuleMatchCount() => void
```

...

Parameters

Example

```

```

JsonParse()

```
JsonParse(string) => JSON object
```

Parse input `string` to JSON object

Parameters

string	JSON as a string.

Return

JSON object if parsing was sucessfull or null

map()

```
map(table, key, default=null) => String
```

Lookup value from mapping table.

Parameters

table	Lookup value from mapping table.
key	String key to lookup.
default	Optional default value.

Return

When key or table is not found, return `null` or default value if provided.

Example

```

```

mapList()

```
mapList(table, list, separator, default) => String
```

Lookup multiple keys (separated by user-defined separator) from mapping table. Result string is joined using the same separator.

Parameters

table	name or ID of the mapping table.
list	string of keys separated by `separator`.
separator	separator used to split `list` and to produce output.

Example

```
>>> mapList("table1", "10,20,30", ",")
value1,value2,value3
```

random()

```
random() => void
```

…

Parameters

Example

```
```

ReadPersistentStorage()

```
ReadPersistentStorage(key) => String
```

Read value from global persistent key-value storage.

Parameters

key	String record key to lookup.

Return

Value referenced by `key` or `null` if key not exist.

Example

```
>>> ReadPersistentStorage("key1")
value1
>>> ReadPersistentStorage("key2")
null
>>> WritePersistentStorage("key2", "value2")
>>> ReadPersistentStorage("key2")
value2
```

SecondsToUptime()

```
SecondsToUptime() => void
```

…

Parameters

Example

```
```

sleep()

```
sleep() => void
```

…

Parameters

Example

```
```

sys()

```
sys() => void
```

…

Parameters

Example

```
```

trace()

```
trace(debugLevel, message) => void
```

Writes `message` to NetXMS main log if current debug level is equal or higher than `debugLevel`.

Parameters

debugLevel	Target debug level.
message	String to be written.

Example

```
>>> trace(0, "Test");
```

typeof()

```
typeof(instance) => string
```

Return type of the `instance`.

Parameters

instance	Instance of the object or primitive.

Return

Name of the type.

Example

```
>>> typeof(1)
int32
>>> typeof(1L)
int64
>>> typeof(%(1, 2, 3))
array
>>> typeof(new Table())
object
>>> typeof(null)
null
```

WritePersistentStorage()

```
WritePersistentStorage(key, value) => void
```

Create or update value in global persistent key-value store.

Parameters

key	String key.
value	String value to be saved.

Example

```
>>> WritePersistentStorage("key1", "value1")
>>> ReadPersistentStorage("key1")
value1
```

weierstrass()

```
weierstrass() => void
```

...

Parameters

Example

```
```

Situations

FindSituation()

```
FindSituation() => void
```

...

Parameters

Example

```
```

GetSituationAttribute()

```
GetSituationAttribute() => void
```

...

Parameters

Example

```
```

Hashes and encoding

Base64Decode()

```
Base64Decode() => void
```

...

Parameters

Example

```
```

Base64Encode()

```
Base64Encode() => void
```

...

Parameters

Example

```
```

md5()

```
md5() => void
```

...

Parameters

Example

```
```

sha1()

```
sha1() => void
```

…

Parameters

Example

```
```

sha256()

```
sha256() => void
```

…

Parameters

Example

```
```

SNMP functions

CreateSNMPTransport()

❗ **This function is deprecated starting from version 3.0.** Please use createSNMPTransport function in Node.

```
CreateSNMPTransport(node, port, context) => SNMP_Transport
```

Create SNMP transport with communication settings defined on the node.

Parameters

node	Target node.
port	Optional parameter with port.
context	Optional parameter with context as a string.

Return

Instance of SNMP_Transport or `null` if failed.

Example

```
>>> transport = CreateSNMPTransport(FindObject("Server1"))
>>> print transport->snmpVersion
2c
```

SNMPGet()

This function is deprecated starting from version 3.0. Please use get function in SNMP_Transport.

```
SNMPGet(transport, oid) => SNMP_VarBind
```

Perform SNMP GET request for `oid` over provided transport.

Parameters

transport	Transport created by CreateSNMPTransport().
oid	SNMP OID string.

Return

Instance of SNMP_VarBind or `null` on failure.

Example

```
>>> transport = CreateSNMPTransport(FindObject("Server1"));
>>> if (transport != null) {
>>>   oid = ".1.3.6.1.2.1.25.1.6.0"; // number of running processes
>>>   varbind = SNMPGet(transport, oid);
>>>   if (varbind != null) {
>>>     trace(1, varbind->name . "=" . varbind->value);
>>>   }
>>>   else {
>>>     trace(0, "SNMPGet() failed");
>>>   }
>>> }
```

SNMPGetValue()

❗ **This function is deprecated starting from version 3.0.** Please use getValue function in SNMP_Transport.

```
SNMPGetValue(transport, oid) => String
```

Perform SNMP GET request for `oid` over provided transport and return single string value instead of varbind.

This function is a wrapper for SNMPGet().

Parameters

transport	Transport created by CreateSNMPTransport().
oid	SNMP OID string.

Return

String value of the result or `null` on failure.

Example

```
>>> transport = CreateSNMPTransport(FindObject("Server1"));
>>> if (transport != null) {
>>> oid = ".1.3.6.1.2.1.25.1.6.0"; // number of running processes
>>> value = SNMPGetValue(transport, oid);
>>> if (value != null) {
>>>     trace(1, value);
>>> }
>>> else {
>>>     trace(0, "SNMPGetValue() failed");
>>> }
>>> }
```

SNMPSet()

❗ **This function is deprecated starting from version 3.0.** Please use set function in SNMP_Transport.

```
SNMPSet(transport, oid, value, dataType) => Boolean
```

Perform SNMP SET request for `oid` over provided transport. Return boolean success indicator. `value` is automatically converted from string based in `dataType`. If `dataType` is not provided, default type

"STRING" will be used.

Parameters

transport	Transport created by CreateSNMPTransport().
oid	SNMP OID string.
value	New value.
dataType	Type of the `value`, default to "STRING". <<SNMP data types

Return

Boolean. TRUE on success and FALSE in case of failure.

Example

```
>>> if (!SNMPSet(transport, oid, "192.168.0.1", "IPADDR") {
>>>     trace(1, "SNMPSet failed");
>>> }
```

SNMPWalk()

This function is deprecated starting from version 3.0. Please use set function in SNMP_Transport.

```
SNMPWalk(transport, oid) => Array
```

Perform SNMP WALK request for `oid` over provided transport and return collected values as array of SNMP_VarBind or `null` on failure.

Parameters

transport	Transport created by CreateSNMPTransport().
oid	SNMP OID string.

Return

Array of SNMP_VarBind or `null` or failure.

Example

```
>>> transport = CreateSNMPTransport(FindObject("Server1"));
>>> if (transport != null) {
>>>     oid = ".1.3.6.1.2.1.25.4.2.1.2"; // Names of the running processes
>>>     vars = SNMPWalk(transport, oid);
>>>     if (vars != null) {
>>>         foreach (v: vars) {
>>>             trace(1, v->name."=".v->value);
>>>         }
>>>     }
>>> }
```

Filesystem functions

I/O functions are disabled by default. To enable, please modify server parameter "NXSL.EnableFileIOFunctions".

CopyFile()

```
CopyFile() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```

```

CreateDirectory()

```
CreateDirectory() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```

```

DeleteFile()

```
DeleteFile() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```
```

FileAccess()

```
FileAccess() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```
```

OpenFile()

```
OpenFile() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```
```

RemoveDirectory()

```
RemoveDirectory() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```
```

RenameFile()

```
RenameFile() => void
```

AF_ENABLE_NXSL_FILE_IO_FUNCTIONS

…

Parameters

Example

```
```

Class Reference

Access point

Represents NetXMS access point object.

Instance attributes

`icmpAverageRTT ⇒ Integer`
: ICMP average response time for primary address. Will return null if no information.

`icmpLastRTT ⇒ Integer`
: ICMP last response time for primary address. Will return null if no information.

`icmpMaxRTT ⇒ Integer`
: ICMP maximal response time for primary address. Will return null if no information.

`icmpMinRTT ⇒ Integer`
: ICMP minimal response time for primary address. Will return null if no information.

`icmpPacketLoss ⇒ Integer`
: ICMP packet loss for primary address. Will return null if no information.

`index ⇒ Integer`
: Index

`model ⇒ String`
: Model

`node ⇒ Node`
: Parent node

`serialNumber ⇒ String`
: Serial number

`state ⇒ String`
: State from Access point state

`vendor ⇒ String`
: Vendor

Constants

Access point state

Description	Value
AP_ADOPTED	0
AP_UNADOPTED	1

Description	Value
AP_DOWN	2
AP_UNKNOWN	3

Alarm

Represents NetXMS alarm.

Instance attributes

`ackBy` ⇒ `Number`
    ID of user who acknowledged this alarm.

`creationTime` ⇒ `Number`
    Unix timestamp of the alarm creation time.

`dciId` ⇒ `Number`
    If alarm was created as a result of DCI threshold violation, this attribute will contain ID of the DCI.

`eventCode` ⇒ `Number`
    Event code of originating event.

`eventId` ⇒ `Number`
    ID of originating event.

`eventTagList` ⇒ `Number`
    List of event tags as a coma separated string

`helpdeskReference` ⇒ `String`
    Helpdesk system reference (e.g. issue ID).

`helpdeskState` ⇒ `Number`
    Helpdesk state:

- 0 = Ignored
- 1 = Open
- 2 = Closed

`id` ⇒ `Number`
    Unique identifier of the alarm.

`key` ⇒ `String`
    Alarm key.

`lastChangeTime` ⇒ `Number`
    Unix timestamp of the last update.

`message ⇒ String`
: Alarm message.

`originalSeverity ⇒ Number`
: Original severity of the alarm.

`repeatCount ⇒ Number`
: Repeat count.

`resolvedBy ⇒ Number`
: ID of user who resolved this alarm.

`ruleGuid ⇒ String`
: Guid of the rule that generated the event.

`severity ⇒ Number`
: Current alarm severity.

`sourceObject ⇒ Number`
: ID of the object where alarm is raised.

`state ⇒ Number`
: Alarm state:

  - 0 = Outstanding
  - 1 = Acknowledged
  - 2 = Resolved
  - 17 = Sticky acknowledged

Instance methods

`acknowledge() ⇒ Number`
: Acknowledge alarm. Return `0` on success or error code on failure.

`resolve() ⇒ Number`
: Resolve alarm. Return `0` on success or error code on failure.

`terminate() ⇒ Number`
: Terminate alarm. Return `0` on success or error code on failure.

`addComment(commentText, syncWithHelpdesk) ⇒ Number`
: Add new alarm comment.

Parameters

commentText	String	Text of the new alarm comment.
syncWithHelpdesk	String	Optional. If synchronization with helpdesk should be done. TRUE by default.

Return

Id of the newly created alarm comment.

`getComments() ⇒ Array`

Get array of alarm comments.

Return

Array of Alarm comment objects.

Alarm comment

Represents NetXMS alarm comment.

Instance attributes

`id ⇒ Number`

Alarm comment ID.

`changeTime ⇒ Number`

Unix timestamp of the alarm comment last modification time.

`userId ⇒ Number`

ID of user who last modified this alarm commant.

`text ⇒ Number`

Alarm comment text.

Chassis

Instance attributes

`controller`
`controllerId`
`flags`
`rack`
`rackId`
`rackHeight`
`rackPosition`

Cluster

Instance attributes

`nodes`
`zone`
`zoneUIN`

Instance methods

`getResourceOwner(name) ⇒ Node`

Get node which currently owns named resource.

Parameters

name	String	Name of the resource.

Return

Node object instance which currently owns resource of `null` if failed.

Component

Instance attributes

`class ⇒ String`

Type of the component:

- `unknown`
- `chassis`
- `backplane`
- `container`
- `power supply`
- `fan`
- `sensor`
- `module`
- `port`
- `stack`

`children ⇒ Array`

List of direct children (Array of Component object intances).

`description ⇒ String`
`firmware ⇒ String`

Component firmware version, if available.

`ifIndex ⇒ Number`
`model ⇒ String`

Component model number, if available.

`name ⇒ String`

Component name, if available.

`serial ⇒ String`

Component serial number, if available.

`vendor ⇒ String`

Component vendor, if available.

Container

Object represent container, extends NetObj.

Instance attributes

`autoBindScript ⇒ String`
: Source of the script for automatic binding.

`isAutoBindEnabled ⇒ Boolean`
: Indicate if automatic binding is enabled.

`isAutoUnbindEnabled ⇒ Boolean`
: Indicate if automatic unbinding is enabled.

Instance methods

`setAutoBindMode(enableBind, enableUnbind) ⇒ void`
: Set automatic bind mode for the container.

Parameters

enableBind	Boolean	Script should be used for automatic binding.
enableUnbind	Boolean	Script should be used for automatic unbinding.

`setAutoBindScript(script) ⇒ void`
: Update automatic binding script source.

Parameters

script	String	Script source.

DataCollectionTarget

Abstract class that represents any object that can collect data. Extends NetObj.

Instance attributes

`templates ⇒ Array`
: Returns array of templates (NetObj) applied on this object. Return value also affected by trusted nodes settings.

Example

```
// Log names and ids of all accessible templates for current node
templates = $node->templates;
foreach(t : templates)
{
    trace(1, "Template object: name='" . t->name . "' id=" . t->id);
}
```

Instance methods

`readInternalParameter(name) ⇒ String`

Reads server internal object metric (metric with source "Internal").

Parameters

name	String	Metric name.

DCI

Represents Data Collection Item (DCI).

Instance attributes

`activeThresholdSeverity`

Severity of the active threshold. If there are no active thresholds, defaults to `0` (NORMAL).

`comments ⇒ String`

DCI Comments (since 2.0-M5)

`dataType ⇒ Integer`

Data type of the DCI.

`description`

Description

`errorCount`

Number of consecutive data collection errors

`hasActiveThreshold`

`id`

Unique DCI identifier

`instance`

DCI instance (only for single value DCIs)

`instanceData`

`lastPollTime`

Time of last DCI poll (either successful or not) as number of seconds since epoch (1 Jan 1970

00:00:00 UTC)

`name`
: Parameter’s name

`origin`
: Data origin (source); possible values are:

- 0 = Internal
- 1 = NetXMS agent
- 2 = SNMP agent
- 3 = Check Point SNMP agent
- 4 = Push

`relatedObject ⇒ NetObj`
: Related object or null if there is no object

`status`
: DCI status; possible values are:

- 0 = Active
- 1 = Disabled
- 2 = Not supported

`systemTag`
: System tag. Always empty for user-defined DCIs.

`template`
`templateId`
`templateItemId`
`type`
: DCI type:

- 1 = single value
- 2 = table

Instance methods

`forcePoll() ⇒ void`
: Start DCI force poll.

Event

Represents NetXMS event object.

Instance attributes

`code` ⇒ `Number`

Event code

`customMessage` ⇒ `String`

Custom message set in event processing policy by calling `setMessage`. Get/set attribute.

`dci` ⇒ `DCI`

DCI object of class DCI that is source for this event or NULL if generated not by threshold

`dci` ⇒ `Number`

DCI id that is source for this event or 0 if generated not by threshold

`id` ⇒ `Number`

Unique event identifier.

`message` ⇒ `String`

Event message. Get/set attribute.

`name` ⇒ `String`

Event name.

`origin` ⇒ `Number`

Origin of the event

- 0 - SYSTEM
- 1 - AGENT
- 2 - CLIENT
- 3 - SYSLOG
- 4 - SNMP
- 5 - NXSL
- 6 - REMOTE_SERVER

`originTimestamp` ⇒ `Number`

The time when the event was generated in the origin.

`parameters` ⇒ `Array`

List of event parameters. Starting index is 1.

`parameterNames` ⇒ `Array`

List of named event parameters (e.g. "dciId"), which can be accessed by `object->parameterName`.

`severity` ⇒ `Number`

Event severity code. Get/set attribute.

`source` ⇒ `NetObj`

Source object (inherited from NetObj, exact type can be checked with classof() function) for the event.

`sourceId ⇒ Number`

ID of the source object for the event.

`tags ⇒ Array`

Event tags as an array of strings.

`tagList ⇒ String`

Event tags as a coma separated list.

`timestamp`

Unix timestamp of the event.

`$1⋯$n`

Shortcut for `parameters[n]` (e.g. "$event→parameters[3]" can be replaced with "$event→$3").

`$⋯`

Named event parameters can be accessed directly by the name (e.g `$event->dciId`). List of available named parameters can be accessed with `parameterNames` attribute. Get/set attribute.

Instance methods

`setMessage(message) ⇒ void`

Set event message to `message`.

Parameters

message	Message string

`setSeverity(severityCode) ⇒ void`

Change event severity to `severityCode`.

Parameters

severityCode	Numeric severity code: • 0 - NORMAL • 1 - WARNING • 2 - MINOR • 3 - MAJOR • 4 - CRITICAL

`hasTag(tag) ⇒ Boolean`

Return if event has specific tag.

Parameters

tag	String tag

`addParameter(name, value) ⇒ void`

Set event parameter

Parameters

name	String	Parameter name. Optional parameter.
value	String	Parameter value.

`toJson() ⇒ String`

Serialize object to JSON.

Return

String representation of the object in JSON format.

`addTag(tag) ⇒ void`

Set event tag, which can be later accessed via `tags` attribute.

Parameters

tag	String tag

`corelateTo(eventId) ⇒ void`

Sets root cause id for the event

Parameters

eventId	Root cause event id

`expandString(String) ⇒ String`

Expand string, by replacing macros.

Parameters

String	String to expand

Return

Formated string

`removeTag(tag) ⇒ void`

Remove tag form event tag list

Parameters

tag	String tag

`toJson() ⇒ String`

Serialize object to JSON.

Return

String representation of the object in JSON format.

FILE

Instance attributes

`eof ⇒ ?`
`name ⇒ ?`

Instance methods

`close() ⇒ ?`
`read() ⇒ ?`
`readLine() ⇒ ?`
`write() ⇒ ?`
`writeLine() ⇒ ?`

GeoLocation

Represents geographical location (defined by latitude and longitude).

Instance attributes

`isManual ⇒ ?`
`isValid → ?`
`latitude ⇒ Number`
Latitude as floating point number

`latitudeText ⇒ String`
Latitude as text

`longitude ⇒ Number`
Longitude as floating point number

`longitudeText ⇒ String`
Longitude as text

`type ⇒ Number`
Data source type:

- 0 – Unset
- 1 – Manual
- 2 - GPS
- 3 - Network

Constructors

`GeoLocation(latitude, longitude, type=1)`
Create instance of the class based on floating-point `latitude` and `longitude`. Optional argument `type` can be used to override default value 1 ("Manual").

Constants

Location Types

Value	Description
0	Unset (represents unknown location)
1	Manual (set by system administrator)
2	Automatic (obtained from GPS)
3	Network (obtained from network, for example using WiFi AP database)

Examples

Print node location

```
>>> nodeLoc = $node->geolocation
>>> println(nodeLoc->latitudeText)
N 48° 00' 0.000"
>>> println(nodeLoc->longitudeText)
E 22° 00' 0.000"
```

Set node location

```
>>> nodeLoc = GeoLocation(22.11, 48.12, 1)
>>> $node->setGeoLocation(nodeLoc)
>>> println($node->geolocation->latitudeText)
N 48° 12' 0.000"
>>> println($node->geolocation->longitudeText)
E 22° 11' 0.000"
```

Clear location

```
>>> $node->clearGeoLocation()
>>> println($node->geolocation)
null
```

InetAddress

Instance attributes

`address` → ?
`family` → ?
`isAnyLocal` → ?
`isBroadcast` → ?
`isLinkLocal` → ?
`isLoopback` → ?
`isMulticast` → ?
`isValid` → ?
`isValidUnicast` → ?
`mask` → ?

Constructors

`InetAddress()`
`InetAddress(string)`

Interface

Represent interface object. Inherit all attributes and methods of the NetObj class.

Constants

Interface states

Code	Description
0	Unknown
1	Up
2	Down
3	Testing

Interface expected states

Code	Description
0	Up
1	Down
2	Ignore

Instance attributes

`adminState`
: Administrative state of the interface.

`alias`
: Interface alias (usually value of SNMP ifAlias).

`bridgePortNumber`
: Bridge port number for this interface.

`chassis`
: Parent chassis

`description`
: Interface description

`dot1xBackendAuthState`
: 802.1x back-end authentication state

`dot1xPaeAuthState`
: 802.1x PAE authentication state

`expectedState`
: Expected state of the interface.

`flags` Interface flags (bit mask, `uint32`).

`icmpAverageRTT ⇒ Integer`
: ICMP average response time for current interface. Will return null if no information.

`icmpLastRTT ⇒ Integer`
: ICMP last response time for current interface. Will return null if no information.

`icmpMaxRTT ⇒ Integer`
: ICMP maximal response time for current interface. Will return null if no information.

`icmpMinRTT ⇒ Integer`
: ICMP minimal response time for current interface. Will return null if no information.

`icmpPacketLoss ⇒ Integer`
: ICMP packet loss for current interface. Will return null if no information.

`ifIndex`
: Interface index.

`ifType`
: Interface type

`ipAddressList → ?`
`ipNetMask`
: IP network mask (number of bits).

`isExcludedFromTopology`
: `TRUE` if this interface excluded from network topology

`isIncludedInIcmpPoll`
: `TRUE` if this interface is included in ICMP statistics

`isLoopback`
: `TRUE` if this interface is a loopback

`isManuallyCreated`
: `TRUE` if this interface object was created manually by NetXMS administrator

`isPhysicalPort`
: `TRUE` if this interface object represents physical port

`macAddr`
: String representation of MAC address separated by ":".

`module`
: Module

`mtu`
: Interface MTU (0 if unknown).

`node`
: Parent node object

`operState`
: Operational state.

`peerInterface`
: Peer interface object if known, otherwise `null`.

`peerNode`
: Peer node object if known, otherwise `null`.

`pic ⇒ Integer`
: Phisical location.

`port= > Integer`
: Port number.

`speed`
: Speed of the interface.

`vlans → ?`

`zone`
: Zone object (null if zoning is disabled).

`zoneUIN → Integer`
: Zone UIN of this interface.

Instance methods

`setExcludeFromTopology(excluded) ⇒ void`
: Change `isExcludedFromTopology` flag.

Parameters

excluded	Boolean	`TRUE` if interface should be excluded.

`setExpectedState(newState) ⇒ void`
: Set expected state to `newState`.

Parameters

newState	Number	New state as defined by Interface expected states.

`setIncludeInIcmpPoll(enabled) ⇒ void`
Enabele/Disable ICMP statistics collection for current interface.

Parameters

enabled	Boolean	If this interface should be included in ICMP statistics.

JSON array

Represents JSON object

Instance methods

`append(value) ⇒ void`
Appends value to JSON array.

`get(index) ⇒ ?`
Retruns aarray value by the index. Value type depensd on the type used in JSON.

Return

Arrtubute value

`insert(index, value) ⇒ void`
Sets value to the provided index in JSON array, moving existing element in this position

`serialize() ⇒ String`
Returns string with serialized JSON

`set(index, value) ⇒ void`
Sets value to the provided index in JSON array, replasing existing element in this position

Return

String with JSON

Constructors

`JsonArray()`
Creates new JSON array.

JSON object

Represents JSON object

Instance attributes

Attribute values can be accessed in the same way as instance attribute.

Instance methods

`get(key) ⇒ ?`
: Retruns attribute value by the key. Value type depensd on the type used in JSON.

Return

Arrtubute value

`keys() ⇒ Array`
: Returns attribute array

Return

Arrtubute array

`serialize() ⇒ String`
: Returns string with serialized JSON

Return

String with JSON

`set(key, value) ⇒ void`
: Sets attribute referenced by key to the given value.

Constructors

`JsonObject()`
: Creates new JSON object.

MobileDevice

Instance attributes

`batteryLevel → ?`
`deviceId → ?`
`model → ?`
`osName → ?`
`osVersion → ?`
`serialNumber → ?`
`userId → ?`
`vendor → ?`

NetObj

Base class for all NetXMS objects.

Instance attributes

Object custom attributes can be accessed in the same way as instance attribute. If name of the custom attribute overlaps with the instance attribute, method NetObj::getCustomAttribute() should be used instead.

`alarms ⇒ array`

List of active Alarms for this object.

`backupZoneProxy ⇒ Node`

Currently selected backup zone proxy (`null` if zoning is disabled or backup proxy is not assigned)

`backupZoneProxyId ⇒ Integer`

ID of currently selected backup zone proxy (`0` if zoning is disabled or backup proxy is not assigned)

`children ⇒ array`

List of child objects (inherited from NetObj). Use classof() to differentiate.

`city ⇒ String`

Postal address - city.

`comments ⇒ String`

Object comments.

`country ⇒ String`

Postal address – country.

`customAttributes`

Hash map of object custom attributes.

`geolocation ⇒ GeoLocation`

Object geographical location.

`guid ⇒ String`

Object GUID as `string`.

`id ⇒ Integer`

Unique object identifier.

`ipAddr ⇒ String`

Primary IP address.

`isInMaintenanceMode ⇒ Boolean`

Maintenance mode indicator (`true` if object currently is in maintenace mode).

`mapImage ⇒ String`

GUID of object image used for representation on the maps.

`name ⇒ String`

Object name.

`parents ⇒ array`
: List of direct parents for this object (inherited from NetObj, most likely either Container or Cluster).

`postcode ⇒ String`
: Postal address – postal code.

`primaryZoneProxy ⇒ Node`
: currently selected primary zone proxy (`null` if zoning is disabled or primary proxy is not assigned)

`primaryZoneProxyId ⇒ Integer`
: ID of currently selected primary zone proxy (`0` if zoning is disabled or primary proxy is not assigned)

`status ⇒ Integer`
: Current object status.

`streetAddress ⇒ String`
: Postal address – street.

`type ⇒ Integer`
: Object type.

Instance methods

`bind(childObject) ⇒ void`

This method is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

Bind `childObject` to the current object as a child.

Parameters

object	NetObj	Object to bind as a child to the current object.

`bindTo(parentObject) ⇒ void`

This method is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

Bind current object to `parentObject` as a child.

Parameters

object	NetObj	Object to bind as a parent to the current object.

`clearGeoLocation() ⇒ void`
: Clears GeoLocation data from the node

`delete() ⇒ void`

Deletes current object.

`deleteCustomAttribute(name) ⇒ void`

Delete custom attribute.

Parameters

name	String	Name of the attribute to delete.

`enterMaintenance() ⇒ void`

Enable maintenance mode for the object.

`getCustomAttribute(name) ⇒ String`

Returns value of the custom attribute with the provided name.

Parameters

name	String	Name of the attribute to get value form.

`leaveMaintenance() ⇒ void`

Disable maintenance mode for the object.

`manage() ⇒ void`

Sets object to managed state. Has no affect if object already managed.

`rename(name) ⇒ void`

Rename object.

Parameters

name	String	New object name

`setComments(comment) ⇒ void`

Set object comments

Parameters

comment	String	Comment to be set

`setCustomAttribute(key, value, inherit=false) ⇒ void`

Update or create custom attribute with the given key and value.

Parameters

key	String	Attribute key
value	String	Attribute value

inherit	Boolean	Optional parameter. If not set - inheritance will not be changed. `true` to inherit, `false` not to inherit.

`setGeoLocation(newLocation) ⇒ void`

Sets node geographical location.

Parameters

newLocation	GeoLocation

`setMapImage(image) ⇒ void`

Sets object image, that will be used to display object on network map

Parameters

image	String	GUID or name of image from image library

`setStatusCalculation(type, …) ⇒ void`

Sets status calculation method.

Parameters

type	Integer	Status calculation type. One of Status callculation types
…	Integer(s)	If single threshold or multiple thresholds type is selected, then threshold or thresholds in percentage should be provided as next parameters.

`setStatusPropagation(type, …) ⇒ void`

Sets status propagation method.

Parameters

type	Integer	Status propagation type. One of Status propagation types
…	Integer(s)	For fixed value type - value (Object Statuses) should be provided. For relative - offset should be provided. For severity - severity mapping should be provided (4 numbers Object Statuses).

`unbind(object) ⇒ void`

This method is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

Unbind provided object from the current object.

Parameters

object	NetObj	Object to unbind from the current object.

`unbindFrom(object) ⇒ void`

> This method is disabled by default and should be explicitly enabled by setting configuration parameter "NXSL.EnableContainerFunctions".

Unbind current object from the provided object.

Parameters

object	NetObj	Object to unbind from the current object.

`unmanage() ⇒ void`

Set object into unmanaged state. Has no effect if object is already in unmanaged state.

Constants

Object Statuses

Code	Description
0	Normal
1	Warning
2	Minor
3	Major
4	Critical
5	Unknown
6	Unmanaged
7	Disabled
8	Testing

Object Types

Code	Description
0	Generic
1	Subnet

Code	Description
2	Node
3	Interface
4	Network
5	Container
6	Zone
7	Service Root
8	Template
9	Template Group
10	Template Root
11	Network Service
12	VPN Connector
13	Condition
14	Cluster

Status callculation types

Code	Description
0	Default
1	Most critical
2	Single threshold
3	Multiple thresholds

Status propagation types

Code	Description
0	Default

Code	Description
1	Unchanged
2	Fixed
3	Relative
4	Translated

NewNode

Represents newly discovered node object. Used by discovery filters.

Instance attributes

`agentVersion`
: NetXMS agent version string, if available.

`ipAddr`
: String representation of IP address.

`isAgent`
: `TRUE` if NetXMS agent is detected on node.

`isBridge`
: `TRUE` if node is a bridge.

`isCDP`
: `TRUE` if node supports CDP (Cisco Discovery Protocol).

`isLLDP`
: `TRUE` if node supports LLDP (Link Layer Discovery Protocol).

`isPrinter`
: `TRUE` if node is a printer.

`isRouter`
: `TRUE` if node is a router (has IP forwarding enabled).

`isSNMP`
: `TRUE` if SNMP agent detected on node.

`isSONMP`
: `TRUE` if node supports SONMP/NDP (Synoptics/Nortel Discovery Protocol).

`netMask`
: Number of bits in IP network mask.

`platformName`
: Platform name reported by NetXMS agent.

`snmpOID`
: SNMP object identifier (value of .1.3.6.1.2.1.1.2.0).

`snmpVersion`
: Detected SNMP version:

  - 0 = SNMP version 1
  - 1 = SNMP version 2c
  - 2 = SNMP version 3

`subnet`
: IP subnet address as string. Example: node with IP address 192.168.2.7 and network mask 255.255.255.0, is in a subnet 192.168.2.0.

`zone ⇒ ?`
`zoneUIN ⇒ ?`

Node

Represents NetXMS node object. Extends DataCollectionTarget.

Instance attributes

`agentCertificateSubject ⇒ String`
: Subject of certificate issued for agent tunnel on this node.

`agentId ⇒ String`
: NetXMS agent unique ID (`string` representation of GUID). Will return all zeroes GUID if agent is not detected on node or does not have unique ID.

`agentVersion`
: NetXMS agent version as `string`.

`bootTime`
: Number of seconds since node start or `0` if unknown.

`bridgeBaseAddress`
: Base address of the switch formatted as 12 character `string` without separators. Value is only valid for bridges and switches. Special value `000000000000` indicates that address is unknown.

`capabilities ⇒ Integer`
: Detected node capabilities ("Have Agent", "Support SNMP", etc.) Bitwise AND of Node capability flags constants.

`components ⇒ String`
`dependentNodes ⇒ ?`
`driver`
: Named of selected device-specific SNMP driver.

`flags` ⇒ `Integer`
: Bit mask of Node flags.

`hasAgentIfXCounters` ⇒ `Boolean`
: `TRUE` if agent supports 64-bit interface counters.

`hasEntityMIB` ⇒ `Boolean`
: `TRUE` if supports ENTITY-MIB.

`hasIfXTable` ⇒ `Boolean`
: `TRUE` if supports ifXTable.

`hasUserAgent` ⇒ `Boolean`
: `TRUE` if has user agent

`hasVLANs` ⇒ `Boolean`
: `TRUE` if VLAN information available.

`hasWinPDH` ⇒ `Boolean`
: `TRUE` if node supports Windows PDH parameters.

`hypervisorInfo` ⇒ `String`
: Additional information about hypervisor for this node.

`hypervisorType` ⇒ `String`
: Hypervisor type as `string` (usually hypervisor vendor or product name, like VMWare or XEN).

`icmpAverageRTT` ⇒ `Integer`
: ICMP average response time for primary address. Will return null if no information.

`icmpLastRTT` ⇒ `Integer`
: ICMP last response time for primary address. Will return null if no information.

`icmpMaxRTT` ⇒ `Integer`
: ICMP maximal response time for primary address. Will return null if no information.

`icmpMinRTT` ⇒ `Integer`
: ICMP minimal response time for primary address. Will return null if no information.

`icmpPacketLoss` ⇒ `Integer`
: ICMP packet loss for primary address. Will return null if no information.

`interfaces` ⇒ `Array`
: Array with Interface objects, that are under this node. First object placed at index 0.

`is802_1x` ⇒ `Boolean`
: `TRUE` if node supports 802.1x. Equivalent of `isPAE`.

`isAgent` ⇒ `Boolean`
: `TRUE` if NetXMS agent detected on node

`isBridge` ⇒ `Boolean`
: `TRUE` if node is a bridge

`isCDP` ⇒ `Boolean`
: `TRUE` if node supports CDP (Cisco Discovery Protocol)

`isLLDP` ⇒ `Boolean`
: `TRUE` if node supports LLDP (Link Layer Discovery Protocol)

`isLocalManagement` ⇒ `Boolean`
: `TRUE` if node is a local management server (NetXMS server)

`isLocalMgmt` ⇒ `Boolean`
: `TRUE` if node is a local management server (NetXMS server)

`isNDP` ⇒ `Boolean`
: `TRUE` if node supports OSPF/NDP. Equivalent of `isOSPF`.

`isOSPF` ⇒ `Boolean`
: `TRUE` if node supports OSPF/NDP. Equivalent of `isNDP`.

`isPAE`
: `TRUE` if node supports 802.1x. Equivalent of `is802_1x`.

`isPrinter`
: `TRUE` if node is a printer

`isRouter`
: `TRUE` if node is a router (has IP forwarding enabled)

`isSNMP`
: `TRUE` if SNMP agent detected on node

`isSONMP`
: `TRUE` if node supports SONMP/NDP (Synoptics/Nortel Discovery Protocol)

`isSTP` ⇒ `?`

`isUserAgentInstalled` ⇒ `Boolean`
: `TURE` if user agent is installed.

`isVirtual` ⇒ `?`

`isVRRP` ⇒ `Boolean` `TURE` if VRRP supported.

`lastAgentCommTime` ⇒ `Integer`
: Unix timestamp of last time when communication with agent was

`nodeSubType` ⇒ `?`

`nodeType` ⇒ `?`

`platformName` ⇒ `String`
: Platform name reported by NetXMS agent

`rack ⇒ ?`

`rackHeight ⇒ ?`

`rackId ⇒ ?`

`rackPosition ⇒ ?`

`runtimeFlags`

Bit mask of Node runtime flags, `uint32`.

`snmpOID`

SNMP object identifier (result of `.1.3.6.1.2.1.1.2.0` request)

`snmpSysContact`

SNMP system contact (result of `.1.3.6.1.2.1.1.4.0` request)

`snmpSysLocation`

SNMP system location (result of `.1.3.6.1.2.1.1.6.0` request)

`snmpSysName`

SNMP system name (result of `.1.3.6.1.2.1.1.5.0` request)

`snmpVersion`

Configured SNMP version:

- 0: SNMP version 1
- 1: SNMP version 2c
- 2: SNMP version 3

`sysDescription`

System description (value of `System.Uname` for nodes with agents or `.1.3.6.1.2.1.1.1.0` for SNMP nodes)

`vlans ⇒ Array`

Array with object VLAN objects (`null` if there are no VLANs)

`zone ⇒ Zone`

Zone object (`null` if zoning is disabled)

`zoneProxyAssignments ⇒ Integer`

Number of objects where this node is selected as either primary or backup zone proxy (`0` if zoning is disabled or this node is not a zone proxy).

`zoneProxyStatus ⇒ Boolean`

Status of this node as zone proxy (`true` if active).

`zoneUIN ⇒ Integer`

This node zone UIN

Instance methods

`void createSNMPTransport(port, context) ⇒ SNMP_Transport`

Create SNMP transport object of class SNMP_Transport with communication settings defined on the node.

Parameters

port	Integer	Optional parameter with port.
context	String	Optional parameter with context.

`void enableAgent(flag)`

Enable or disable usage of NetXMS agent for all polls.

Parameters

flag	Boolean	If agent usage should be enabled.

`void enableConfigurationPolling(flag) ⇒ void`

Enable or disable configuration polling for a node

Parameters

flag	Boolean	If configuration polling should be enabled.

`enableDiscoveryPolling(flag) ⇒ void`

Enable or disable discovery polling.

Parameters

flag	Boolean	If discovery polling should be enabled.

`enableIcmp(flag) ⇒ void`

Enable or disable usage of ICMP pings for status polls.

Parameters

flag	Boolean	If ICMP pings should be enabled.

`enablePrimaryIPPing(flag) ⇒ void`

Enable or disable usage of ICMP ping for primary IP.

Parameters

flag	Boolean	If primary IP ping should be enabled.

`enableRoutingTablePolling(flag) ⇒ void`

Enable or disable routing table polling.

Parameters

flag	Boolean	If routing table polls should be enabled.

`enableSnmp(flag) ⇒ void`

Enable or disable usage of SNMP for all polls.

Parameters

flag	Boolean	If SNMP communication should be enabled.

`enableStatusPolling(flag) ⇒ void`

Enable or disable status polling for a node.

Parameters

flag	Boolean	If status polls should be enabled.

`enableTopologyPolling(flag) ⇒ void`

Enable or disable topology polling.

Parameters

flag	Boolean	If topology polls should be enabled.

`executeSSHCommand(command) ⇒ void`

Execute SSH command on node.

Parameters

command	String	Command to be executed.

`getInterface(ifIndex) ⇒ Interface`

Get interface object by index.

Parameters

ifIndex	Integer	Index of interface.

`getInterfaceName(ifIndex) ⇒ String`

Get interface name by index.

Parameters

ifIndex	Integer	Index of interface.

`readAgentParameter(name) ⇒ String`

Reads current value of agent metric.

Parameters

name	String	Parameter name.

`readAgentList(name) ⇒ Array`

Reads current value of agent list metric and returns array of strings.

Parameters

name	String	List name.

`readAgentTable(name) ⇒ Table`

Reads current value of agent table metric and returns Table.

Parameters

name	String	Table name.

`readDriverParameter(name) ⇒ String`

Request driver-specific metric directly from network device driver (e.g. Rital).

Parameters

name	String	List name.

Constants

Node flags

Description	Value
DCF_DISABLE_STATUS_POLL	0x00000001
DCF_DISABLE_CONF_POLL	0x00000002
DCF_DISABLE_DATA_COLLECT	0x00000004
NF_REMOTE_AGENT	0x00010000
NF_DISABLE_DISCOVERY_POLL	0x00020000
NF_DISABLE_TOPOLOGY_POLL	0x00040000
NF_DISABLE_SNMP	0x00080000
NF_DISABLE_NXCP	0x00100000
NF_DISABLE_ICMP	0x00200000

Description	Value
NF_FORCE_ENCRYPTION	0x00400000
NF_DISABLE_ROUTE_POLL	0x00800000
NF_AGENT_OVER_TUNNEL_ONLY	0x01000000
NF_SNMP_SETTINGS_LOCKED	0x02000000

Node runtime flags

Description	Value
DCDF_QUEUED_FOR_STATUS_POLL	0x00000001
DCDF_QUEUED_FOR_CONFIGURATION_POLL	0x00000002
DCDF_QUEUED_FOR_INSTANCE_POLL	0x00000004
DCDF_DELETE_IN_PROGRESS	0x00000008
DCDF_FORCE_STATUS_POLL	0x00000010
DCDF_FORCE_CONFIGURATION_POLL	0x00000020
DCDF_CONFIGURATION_POLL_PASSED	0x00000040
DCDF_CONFIGURATION_POLL_PENDING	0x00000080
NDF_QUEUED_FOR_TOPOLOGY_POLL	0x00010000
NDF_QUEUED_FOR_DISCOVERY_POLL	0x00020000
NDF_QUEUED_FOR_ROUTE_POLL	0x00040000

Description	Value
NDF_RECHECK_CAPABILITIES	0x00080000
NDF_NEW_TUNNEL_BIND	0x00100000

Node capability flags

Value	Description
0x00000001	Node supports SNMP
0x00000002	NetXMS agent detected on the node
0x00000004	Node is network bridge
0x00000008	Node is IP router
0x00000010	Node is management server (NetXMS server itself)
0x00000020	Node is printer
0x00000040	Node supports OSPF
0x00000080	CheckPoint SNMP agent detected on port 260
0x00000100	CDP supported
0x00000200	NDP(SONMP) support detected on the node (Nortel/Synoptics/Bay Networks) topology discovery)
0x00000400	Node supports LLDP
0x00000800	Node supportes VRRP
0x00001000	VLAN information available on the node
0x00002000	802.1x support detected
0x00004000	Spanning Tree (IEEE 802.1d) enabled on node
0x00008000	Node supports ENTITY-MIB
0x00010000	Node supports ifXTable

Value	Description
0x00020000	Agent supports 64-bit interface counters
0x00040000	Node supports Windows PDH parameters
0x00080000	Node is wireless network controller
0x00100000	Node supports SMCLP protocol
0x00200000	Running agent is upgraded to new policy type
0x00400000	User (support) agent is installed

NodeDependency

Instance attributes

`id ⇒ ?`
`isAgentProxy ⇒ ?`
`isDataCollectionSource ⇒ ?`
`isICMPProxy ⇒ ?`
`isSNMPProxy ⇒ ?`
`type ⇒ ?`

SNMP_Transport

Represents SNMP Transport functionality. Objects of this class are typically obtained from nodes that support SNMP. Objects of this class used to access SNMP functions of nodes.

Instance attributes

`snmpVersion`
SNMP version used by the transport. Can be "1", "2c" or "3"

Instance methods

`get(oid) ⇒ SNMP_VarBind`
Get the object value from specific node with SNMP GET request. The node and all SNMP communication details defined by SNMP transport. Will return null on failure.

Parameters

oid	String	SNMP object id.

`getValue(oid) ⇒ String`
Get the object value from specific node with SNMP GET request. The node and all SNMP

communication details defined by SNMP transport. This function is similar to SNMPGet but returns string instead of an SNMP_VarBind object. Will return null on failure.

Parameters

oid	String	SNMP object id.

`set(oid, value, dataType) ⇒ Boolean`

Assign a specific value to the given SNMP object for the node. The node and all SNMP communication details defined by SNMP transport. Will return TRUE on success, FALSE in case of failure.

Parameters

oid	String	SNMP object id.
oid	String	Value to assign to oid.
oid	String	SNMP data type (optional).

`walk(oid) ⇒ Array`

Get an array of the SNMP_VarBind from specific node with SNMP WALK request. The node and all SNMP communication details defined by SNMP transport. Will return null on failure.

Parameters

oid	String	SNMP object id.

Constants

SNMP data types

Description	Value
Integer.	INTEGER
Same as INTEGER.	INT
Octet string.	STRING
Object id.	OID
IP address.	IPADDR
Same as IPADDR.	IP ADDRESS
32-bit counter.	COUNTER32

Description	Value
32-bit unsigned integer.	GAUGE32
Timeticks.	TIMETICKS
64-bit counter.	COUNTER64
32-bit unsigned integer.	UINTEGER32
Same as UINTEGER32.	UINT32

SNMP_VarBind

Represents an SNMP varbind concept in NetXMS. A varbind logically consists of an OID and a value.

Instance attributes

`name`
: Object name (OID string).

`printableValue`
: Object value as a printable string.

`type`
: ASN.1 type.

`value`
: Object value as a string.

`valueAsIp`
: Object value IP address, represented as string.

`valueAsMac`
: Object value as MAC address, represented as string.

Subnet

Instance attributes

`ipNetMask ⇒ ?`
`isSyntheticMask ⇒ ?`
`zone ⇒ ?`
`zoneUIN ⇒ ?`

Instance methods

`method() ⇒ void`
: Description.

Table

Represents table object (usually it’s value of table DCI).

Instance attributes

`columnCount ⇒ Number`
: Number of columns.

`columns ⇒ Array<TableColumn>`
: Array of column definitions.

`rowCount ⇒ Numbers`
: Number of rows.

`rows ⇒ Array<TableRow>`
: Array of rows with data.

`title ⇒ String`
: Title of table.

Instance methods

`addColumn(name, [type], [displayName], [isInstance]) ⇒ Number`
`addRow() ⇒ Number`
`deleteColumn(columnId) ⇒ void`
`deleteRow(rowId) ⇒ void`
`get(rowId, columnId) ⇒ String`
`getColumnIndex(columnName) ⇒ Number`
`getColumnName(columnId) ⇒ String`
`set(rowId, columnId, value) ⇒ void`

TableColumn

Represents table column definition object (used by Table class).

Instance attributes

`dataType`
: Data type

`displayName`
: Display name

`isInstanceColumn`
: `TRUE` if column is marked as instance column

`name`
: Column name

TableRow

Instance attributes

`index ⇒ Number`
`values ⇒ Array<String>`

Instance methods

`get(columnId) ⇒ String`
`set(columnId, value) ⇒ void`

TIME

Class containing a calendar date and time broken down into its components. For convenience, all attributes has aliases to match struct tm provided in libc.

Instance attributes

`sec ⇒ Number`
`tm_sec ⇒ Number`
: Seconds after the minute.

`min ⇒ Number`
`tm_min ⇒ Number`
: Minutes after the hour.

`hour ⇒ Number`
`tm_hour ⇒ Number`
: Hours since midnight.

`mday ⇒ Number`
`tm_mday ⇒ Number`
: Day of the month.

`mon ⇒ Number`
`tm_mon ⇒ Number`
: Months since January.

`year ⇒ Number`
`tm_year ⇒ Number`
: Year.

`yday ⇒ Number`
`tm_yday ⇒ Number`
: Days since January 1.

`wday ⇒ Number`
`tm_wday ⇒ Number`
: Days since Sunday.

`isdst ⇒ Boolean`
`tm_isdst ⇒ Boolean`
: Daylight Saving Time flag.

VLAN

Represents VLAN object.

Instance attributes

`id ⇒ Integer`
: VLAN id.

`name ⇒ String`
: VLAN name.

`interfaces ⇒ Array`
: Interfaces in that VLAN (array of objects of class Interface).

Zone

Represent network zone. Inherit all attributes and methods of the NetObj class.

Instance attributes

Previously available attributes proxyNode and proxyNodeId were deprecated starting from version 3.0.

`proxyNodes ⇒ Array<Node>`
: Array of Node objects that are currently set as proxies for this zone.

`proxyNodeIds ⇒ Array<Integer>`
: Array of integers representing identifiers of node objects that are currently set as proxies for this zone.

`uin ⇒ Integer`
: Zone UIN (Unique Identification Number).

Global Constants

Data types of the DCI class

Constant	Value	Description
DCI::INT32	0	Signed 32 bit integer
DCI::UINT32	1	Unsigner 32 bit integer
DCI::INT64	2	Signer 64 bit integer
DCI::UINT64	3	Unsigned 64 bit integer
DCI::STRING	4	String
DCI::FLOAT	5	Floating point number
DCI::NULL	6	Used internally; should be used in the scripts
DCI::COUNTER32	7	32 bit counter
DCI::COUNTER64	8	64 bit counter

Other constants

NXSL::VERSION

Current server version

NXSL::BUILD_TAG

Current server build tag

Formal Grammar

Grammar

```
script ::=
  module |
  expression

module ::=
  module_component { module_component }

module_component ::=
  function |
  statement_or_block |
  use_statement

use_statement ::=
  use any_identifier ";"

any_identifier ::=
  IDENTIFIER |
  COMPOUND_IDENTFIER

function ::=
  sub IDENTIFIER "(" [ identifier_list ] ")" block

identifier_list ::=
  IDENTIFIER { "," IDENTIFIER }

block ::=
  "{" { statement_or_block } "}"

statement_or_block ::=
  statement |
  block

statement ::=
  expression ";" |
  builtin_statement |
  ";"

builtin_statement ::=
  simple_statement ";" |
  if_statement |
  do_statement |
  while_statement |
  for_statement |
  foreach_statement |
  switch_statement |
  array_statement |
  global_statement |
  break ";"
  continue ";"

simple_statement ::=
  keyword [ expression ]

keyword ::=
  exit |
  print |
  println |
  return

if_statement ::=
  if "(" expression ")" statement_or_block [ else statement_or_block ]

for_statement ::=
  for "(" expression ";" expression ";" expression ")" statement_or_block

foreach_statement ::=
  foreach "(" IDENTIFIER ":" expression ")" statement_or_block

while_statement ::=
  while "(" expression ")" statement_or_block

do_statement ::=
  do statement_or_block while "(" expression ")" ";"

switch_statement ::=
  switch "(" expression ")" "{" case { case } [ default ] "}"

case ::=
  case constant ":" { statement_or_block }

default ::=
  default ":" { statement_or_block }

array_statement ::=
  [ global ] array identifier_list ";"

global_statement ::=
  global global_variable_declaration { "," global_variable_declaration } ";"

global_variable_declaration ::=
  IDENTIFIER [ "=" expression ]

expression ::=
  "(" expression ")" |
  IDENTIFIER "=" expression |
  expression "->" IDENTIFIER |
  "-" expression |
  "!" expression |
  "~" expression |
  inc IDENTIFIER |
  dec IDENTIFIER |
  IDENTIFIER inc |
  IDENTIFIER dec |
  expression "+" expression |
  expression "-" expression |
  expression "*" expression |
  expression "/" expression |
  expression "%" expression |
  expression like expression |
  expression ilike expression |
  expression "~=" expression |
  expression match expression |
  expression imatch expression |
  expression "==" expression |
  expression "!=" expression |
  expression "<" expression |
  expression "<=" expression |
  expression ">" expression |
  expression ">=" expression |
  expression "&" expression |
  expression "|" expression |
  expression "^" expression |
  expression "&&" expression |
  expression "||" expression |
  expression "<<" expression |
  expression ">>" expression |
  expression "." expression |
  expression "?" expression ":" expression |
  operand

operand ::=
  function_call |
  type_cast |
  constant |
  IDENTIFIER

type_cast ::=
  builtin_type "(" expression ")"

builtin_type ::=
  int32 |
  int64 |
  uint32 |
  uint64 |
  real |
  string

function_call ::=
  IDENTIFIER "(" [ expression { "," expression } ] ")"

constant ::=
  STRING |
  INT32 |
  INT64 |
  UINT32 |
  UINT64 |
  REAL |
  NULL
```

Terminal symbols

```
IDENTIFIER ::= [A-Za-z_\$][A-Za-z_\$0-9]*
COMPOUND_IDENTIFIER ::= { IDENTIFIER}(::{ IDENTIFIER})+
INTEGER ::= \-?(0x)?[0-9]+
INT32 ::= INTEGER
INT64 ::= {INTEGER}L
UINT32 ::= {INTEGER}U
UINT64 ::= {INTEGER}(UL|LU)
REAL ::= \-?[0-9]+\.[0-9]+
```