2012-12-27
Revision:1.9
http://jx9.symisc.net/

The Jx9 Programming Language Reference Manual.

Preface

Jx9 is an embeddable scripting language also called extension language designed to support general procedural programming with data description facilities. Jx9 is a dynamically typed programming language based on JSON and implemented as a library. Jx9 is written in ANSI C and should compile and run unmodified in any platform including restricted embedded devices with a C Compiler.

Being an extension language, Jx9 has no notion of a *main* program, it only works *embedded* in a host application. This host program can write and read Jx9 variables and can register C/C++ functions to be called by Jx9 code.

The Jx9 download page includes a simple standalone Jx9 interpreter that allows the user to enter and execute Jx9 files against a Jx9 engine. This utility is available in prebuilt binaries forms or can be compiled from source. You can get a copy of the Jx9 interpreter from the download page.

The Language

This paper describes the lexis, the syntax, and the semantics of the Jx9 programming language. In other words, this paper describes which tokens are valid, how they can be combined, and what their combinations mean.

Lexical conventions

Names (also called identifiers) in Jx9 can be any string (UTF-8 stream) of letters, digits, and underscores, not beginning with a digit. This coincides with the definition of names in

most languages. Identifiers are used to name variables, functions and JSON objects fields.

The following *keywords* are reserved and cannot be used as names:

if	else	elseif	for
foreach	while	do (not used)	switch
static	function	case	print
const	default	as	continue
break	exit	die	import
include	string	bool	boolean
int	integer	float	uplink
class (not used)	object (not used)	array (not used)	return
goto (not used)			

Jx9 is a case-sensitive language: **if** is a reserved word, but **IF** and **If** are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as __JX9__, __DATE__, __TIME__, etc.) are reserved for built-in constants.

Instruction separation

As in C or Perl, Jx9 requires instructions to be terminated with a semicolon ';' at the end of each statement.

Example:

```
print 5+5;
continue;
rand();
```

Comments

Jx9 supports C/C++ (//, /* */) style comments as well Unix shell comments (#).

The "one-line" comment styles only comment to the end of the line.

Example:

```
// One-line C++ comment
# Shell style comment
```

C style or block comments end at the first */ encountered. Make sure you don't nest block style comments. It is easy to make this mistake if you are trying to comment out a large block of code.

Example:

```
/*
 * This is a block comment.
 * Block comments don't nest.
 */
```

Types

Jx9 is a *dynamically typed language*. This means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

Being based on JSON, all types are the one introduced by JSON with the addition of the type resource which is used to pass data (Typically malloc()ed pointers) between the host application and the underlying Jx9 virtual machine.

The following are the basic types: **integer**, **real**, **string**, **boolean**, **JSON objects**, **JSON arrays** and the **resource** type.

> ***Tip**: To check the type and value of an expression, use the dump() function.*
>
> *To get a human-readable representation of a type for debugging, use the gettype() function*

We will study each type in details now:

Boolean

This is the simplest type. A boolean expresses a truth value. It can be either **TRUE** or **FALSE**.

To specify a boolean literal, use the keywords **TRUE** or **FALSE**. Both are case-insensitive.

Example:

```
dump(TRUE); //bool(true)
dump(True);  //bool(true)
dump(FALSE); //bool(false)
dump(False);  //bool(false)
```

Converting to Boolean

To explicitly convert a value to boolean, use the ***(bool)*** or ***(boolean)*** casts. However, in most cases the cast is unnecessary, since a value will be automatically converted if an operator, function or control structure requires a boolean argument.

When converting to boolean, the following values are considered FALSE:

- the boolean FALSE itself
- The NULL type
- the integer 0 (zero)
- the float 0.0 (zero)
- the empty string "" , the string "0" and the string "false"
- a JSON array with zero elements
- a JSON object with zero members

Any other value is considered true even negative numbers and the resource type.

Example:

```
dump((bool) -1);            // bool(true)
dump((bool) "");            // bool(false)
dump((bool) "foo");         // bool(true)
dump((bool) 3.14e5);        // bool(true)
dump((bool)[10,11,13]);  // bool(true)
dump((bool){});             // bool(false)
```

```
dump((bool) "false");       // bool(false)
```

Integers

Integers can be specified in decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2) notation, optionally preceded by a sign (- or +).

To use octal notation, precede the number with a ***0*** (zero). To use hexadecimal notation precede the number with ***0x***. To use binary notation precede the number with ***0b***.

Example:

```
print 0xff;   //255
print 1200; //1200
print 0766; //502
print -2; //-2
print 0b1011001; //89
```

Using BNF:

```
decimal     : [1-9][0-9]*
            | 0

hexadecimal : 0[xX][0-9a-fA-F]+

octal       : 0[0-7]+

binary      : 0b[01]+

integer     : [+-]?decimal
            | [+-]?hexadecimal
            | [+-]?octal
            | [+-]?binary
```

Internally, integers are stored in 8 bytes (64 bit) regardless of the target platform. The integer type can can store integer values between -9223372036854775808 and +9223372036854775807 inclusive.

Integer size can be determined using the built-in constant **JX9_INT_SIZE**, and maximum integer value using the constant **JX9_INT_MAX**.

To explicitly convert a value to integer, use either the ***(int)*** or ***(integer)*** casts. However, in most cases the cast is not needed, since a value will be automatically converted if an operator, function or control structure requires an integer argument.

Real number

Reals also known as floating point numbers or doubles can be specified using any of the following syntax:

```
3.142 or 3.4e2 or 6E-3
```

Using BNF:

```
LNUM        [0-9]+
DNUM        ([0-9]*[\.]{LNUM}) | ({LNUM}[\.][0-9]*)
EXPONENT_DNUM [+-]?(({LNUM} | {DNUM}) [eE][+-]? {LNUM})
```

The size of a float is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (the 64 bit IEEE format).

Floating point numbers have limited precision. Although it depends on the system, Jx9 typically uses the IEEE 754 double precision format, which will give a maximum relative error due to rounding in the order of 1.11e-16. Non elementary arithmetic operations may give larger errors, and, of course, error prorogation must be considered when several operations are compounded.

To explicitly convert a value to real number, use either the ***(float)*** casts.

NULL

Null is the type of the value **NULL**, whose main property is to be different from any other value. It usually represents the absence of a useful value. Both **NULL** and **false** make a condition false.

A variable is considered to be NULL if:

- it has been assigned the constant **NULL**.
- it has not been declared yet.

There is only one value of type NULL, and that is the case sensitive keyword **NULL** or **null**.

Example:

```
$temp = NULL;
dump($temp); //null
```

String

A string is a stream of bytes (i.e. UTF-8 or even binary stream).

> **Note**: It is no problem for a string to become very large. Jx9 imposes no boundary on the size of a string; the only limit is the available memory of the computer on which Jx9 is running.

A string literal can be specified in three different ways:

- Single quoted
- nowdoc
- double quoted

Single quoted string

The simplest way to specify a string is to enclose it in single quotes (the character **'**).

To specify a literal single quote, escape it with a backslash (****). To specify a literal backslash, double it (****). All other instances of backslash will be treated as a literal backslash: this means that the other escape sequences you might be used to, such as **\r** or **\n**, will be output literally as specified rather than having any special meaning.

> **Note**: Unlike the double-quoted string, variables and escape sequences for special characters will *not* be expanded when they occur in single quoted strings.

Example:

```
print 'Hello World!'; //Hello World
print 'It\'s me, Mario:'; //It's me, Mario
//This will not expand a new line nor variable content
$var = 'test';
print 'Value of $var = $var\n'; //Value of $var = $var\n
```

Nowdoc

A nowdoc have the same semantic as a single quoted string except that is used to hold big chunk of text such as unparsed Jx9 code or even binary chunks. The construct is ideal for embedding large blocks of text without the need for escaping. It shares some features in common with the SGML *<![CDATA[ ]]>* construct, in that it declares a block of text which is not for parsing.

A nowdoc is identified with the **<<<** operator. After this operator, an identifier is provided, then a newline. The string itself follows, and then the same identifier again to close the quotation.

The closing identifier *must* begin in the first column of the line. Also, the identifier must follow the same naming rules as any other label in Jx9.

Example:

```
$var = 'test';
$str = <<<EOD
Example of string
spanning multiple lines
using nowdoc syntax.
Value of $var = $var\n
//I'm not a comment
EOD;
print $str;
```

Double quoted string

If the string is enclosed in double-quotes (**"**), Jx9 will interpret more escape sequences for special characters:

Sequence	Description

\n	Linefeed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\f	Form feed
****	Backslash
\$	Dollar sign
\"	Double quote
\'	Single quote
\[0-7]{1,3}	the sequence of characters matching the regular expression is a character in octal notation.
\x[0-9A-Fa-f]{1,2}	the sequence of characters matching the regular expression is a character in hexadecimal notation.
\0	NUL byte.

The most important feature of double-quoted strings is the fact that variable names will be expanded:

If a dollar sign (*$*) is encountered, the parser will greedily take as many tokens as possible to form a valid variable name including JSON object and array members.

Example:

```
$name = 'Dean';
print "Hello, my name is = $name\n"; //Hello, my name is Dean
//Declare a simple JSON object
$person = {name : 'Wolf', age : 27 };
print "Mr $person.name is $person.age years old\n"; //Mr Wolf is 27 years old
//Declare a simple JSON array
$num = [27, 512];
print "$num[0+1] is greater than $num[0]\n"; //512 is greater than 27
```

JSON Arrays

A JSON array is simply an ordered sequence of values which can be any valid Jx9 expression including NULL, JSON objects or even other arrays, comma-separated and enclosed in square brackets.

The syntax is as follows:

[*[expr,expr]* **];**

Example:

```
$numbers = [ 15, 20, 50 << 1 ];
```

```
/*
 Declare a JSON array with two random numbers between 0 and 1024 inclusive using the built-in rand() function.
*/
$rand = [ rand() & 1023, rand() % 1024, (512 * 2) >> 1];
//Dump array contents
print $numbers,JX9_EOL,$rand;
```

To access array values simply use the subscript operator. The notation is:

```
  $array_name[expr];
```

Example:

```
//Declare a JSON array with three fields
$my_array = [ __TIME__, __DATE__, uname() ];
print $my_array[0]; //14:24:05
print JX9_EOL;
print $my_array[1]; //2013-01-06
print JX9_EOL;
print $my_array[2]; //Windows 8...
```

JSON Objects

A JSON object is simply an unordered collection of key:value pairs with the ':' character separating the key and the value (Which can be any valid Jx9 expression including NULL, JSON arrays or even another objects), comma-separated and enclosed in curly braces; the keys must be an identifier or a string and should be distinct from each other.

Example:

```
//Declare a simple JSON object
$user = {
    name : 'John Smith',
    login : 'jms123',
    age   : 25
};
print "username= $user.name;\n name = $user.login\n age = $user.age";
```

This example when running should display:

username = jms123

name = John Smith

age = 25

To access object member, simply use the *member access* operator (Also called dot operator) as follows:

```
$person = {
   firstName : "John",
   lastName: "Smith",
   age: 25,
   addr: {
      "streetAddress": "21 2nd Street",
      city: "New York",
      state : "NY",
      postalCode : 10021
   }
}; //Don't forget the semi-colon here

print $person.addr.city; //New York
print JX9_EOL;
print $person.addr.postalCode; //10021
```

Resource

The type resource or userdata in the Jx9 jargon is a special type which is provided to allow arbitrary C data to be stored in Jx9 variables and returned from foreign function.

The built-in standard library rely heavily on this type and is used for example by the implementation of the IO stream functions such as fopen(), fread(), fwrite() and so forth.

This type corresponds to a block of raw memory and has no pre-defined operations in Jx9, except assignment and identity test.

Type casting

Type casting in Jx9 works much as it does in C. the name of the desired type is written in parentheses before the expression which is to be cast. As of this release, the supported type are:

- Integer ==> **(int)** or **(integer)**
- Boolean ==> **(bool)** or **(boolean)**
- Real ==> **(float)**
- String ==> **(string)**

Forced cast of JSON arrays or objects is not supported and is planned for future version.

Example:

```
$var = (int) 2.3e+1; //Force an int cast
$var = (string)0xff; //force a string cast
```

Jx9 does not require explicit type definition in variable declaration; a variable's type is determined by the context in which the variable is used. That is, if a string value is assigned to variable *$var*, *$var* becomes a string. If an integer value is then assigned to *$var*, it becomes an integer.

An example of Jx9 automatic type conversion is the addition operator '+'. If either operand is a real, then both operands are evaluated as reals, and the result will be a real.

Otherwise, the operands will be interpreted as integers, and the result will also be an integer. Note that this does *not* change the types of the operands themselves; the only change is in how the operands are evaluated and what the type of the expression itself is.

Example:

```
$test = "256";
dump($test); //string(3,'256')
$test += 10;
dump($test); //int(266)
$test += "25 years old";
dump($test); //int(291)
$test *= 2.5;
dump($test); //float(3.59679...)
```

***Tip**: To check the type and value of an expression, use the dump() function.*

Variables

Variables are places that store values. There are three kinds of variables in Jx9: global variables, static variables and local variables.

Variables in Jx9 are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

Variable names (UTF-8 Stream) follow the same rules as other labels in Jx9. Refer to the lexical convention section for additional information.

Example:

```
$var = 'Foo';
$Var = 'Bar';

print "$var, $Var";    // outputs "Foo Bar"


$概要 = "JX9 Scripting Engine";
$文書 = "http://jx9.symisc.net";

/* Test */
dump($概要,$文書);
```

Variable scope

The scope of a variable is the context within which it is defined. For the most part all Jx9 variables only have a single scope. This single scope spans imported files as well. For example:

```
$foo = __TIME__;
import 'driver.jx9';
```

Here the *$foo* variable will be available within the imported *driver.jx9* script.

However, within user-defined functions a local function scope is introduced. Any variable used inside a function is by default limited to the local function scope. For example:

```
$foo = rand();  //A random integer
function test()
{
  print $foo; //null, since $foo is not declared in this scope.
}
```

This script will not produce any output because the print statement refers to a local version of the *$foo* variable which is not declared and assigned a value in this scope. You may notice that this is a little bit different from the C language in that global variables in C are automatically available to functions unless specifically overridden by a local definition.

The *uplink* keyword.

In order to modify or to make a global variable available in a local scope, you must use the **uplink** keyword with the name of the target global variables as follows:

```
$foo = rand();  //A random integer
$bar = rand_str(3);  //A random string of length 3
function test()
{
  uplink $foo, $bar; //Make $foo and $bar available in the current scope
  print $foo,' ',$bar; //Output the old value of $foo and $bar.
  /* Modify $foo and $bar */
  $foo = 1023;
  $bar = 'New value';
  print JX9_EOL;
}
test();
/* Global scope */
print $foo; //1023
print $bar; //New value
```

That is, with the **uplink** statement, all references to either variable will refer to the global version. There is no limit to the number of global variables that can be manipulated by a function.

The *static* keyword.

a **static variable** is a variable that has been allocated statically and whose lifetime extends across the entire run of the program. That is, a static variable exists only in a local function scope, but it does not lose its value when program execution leaves this scope.

To declare a variable static, simply precede it with the keyword **static** as follows:

```
function test()
{
   static $num = rand() & 1023; //Random integer between 0 and 1024 inclusive
```

```
  /* Print the number and increment its value by one */
  print $num++;
  print JX9_EOL;
 }
 test(); //59
 test(); //60
 test(); //61
 test(); //62
```

Note that static variables can take any complex expressions including function calls as their initialization value.

Declaring a variable static in the global scope is a no-op.

Static variables also provide one way to deal with recursive functions. A recursive function is one which calls itself. Care must be taken when writing a recursive function because it is possible to make it recurse indefinitely. You must make sure you have an adequate way of terminating the recursion. The following simple function recursively counts to 5, using the static variable *$count* to know when to stop:

```
function test()
{
  static $count = 0;
  /* Increment the static counter */
  if( $count++ < 5 ){
   /* Recurse and call test() again */
   test();
  }
 }
```

Predefined variables

Jx9 is shipped with a set of predefined global variables available in all scopes (without the need of the uplink statement. The following table summarize the predefined variables and their purposes.

Predefined variable name	Purpose
$argv	JSON array holding command line arguments.
$_ENV	JSON object holding environments info.
$_FILES	JSON array holding the name of the imported files.
$_GET	JSON object holding decoded GET query.
$_POST	JSON object holding decoded POST request.
$_HEADER	JSON object holding HTTP request MIME

	header.

Note that host applications can define their own predefined variables using the jx9_vm_config() interface with a configuration verb set to JX9_VM_CONFIG_CREATE_VAR.

Also note that the predefined variables defined above can also be populated by the host application with the desired *key, value* pair using the jx9_vm_config() interface.

Also, the host application can extract the contents of or more variables declared inside the target Jx9 script using a call to jx9_vm_extract_variable().

Expressions

*An **expression** in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language, which computes and then produces (returns, in a stateful environment) another value. This process, like for mathematical expressions, is called evaluation. The value can be of various types, such as numerical, string, and logical. (src: Wikipedia)*

For example, 2+3 is an arithmetic and programming expression which evaluates to 5. A variable is an expression because it denotes a value in memory, so $y+6 is an expression. An example of a relational expression is 4 != 4, which evaluates to false

Expressions are the most important building stones of Jx9. In Jx9, almost anything you write is an expression. The simplest yet most accurate way to define an expression is "anything that has a value".

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value. For example, the expression $x = 6 is an expression that assigns the variable $x the value seven. This expression itself evaluates to seven. Such expressions use *assignment operators*. On the other hand, the expression 4 + 2 simply evaluates to seven; it does not perform an assignment. The operators used in such expressions are referred to simply as *operators*.

Operators

An operator is something that takes one or more values (or expressions, in programming jargon) and yields another value (so that the construction itself becomes an expression).

Operators can be grouped according to the number of values they take. Unary operators take only one value, for example *!* (the logical not operator) or ++ (the increment operator). Binary operators take two values, such as the familiar arithmetical operators + (plus) and - (minus), and the majority of Jx9 operators fall into this category. Finally, there is a single ternary operator, *?* , which takes three values; this is usually referred to simply as "the ternary operator" (although it could perhaps more properly be called the conditional operator).

Jx9 has the following types of operators. This section describes the operators and contains information about operator precedence.

- Assignment Operators
- Arithmetic Operators

- Comparison Operators
- Ternary Operator
- Bitwise Operators
- Logical Operators
- String Operators
- Member Access Operators
- JSON Arrays/Objects Operators
- Comma Operator

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (**=**), which assigns the value of its right operand to its left operand. That is, $x = 10 assigns the value of 10 to the variable $x.

In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic, JSON array/object union and string operators that allow you to use a value in an expression and then set its value to the result of that expression.

Assignment operators:

Shorthand operator	Meaning	Description
$x **=** *expr*	-	Simple assignment.
$x **+=** *expr*	x = x + y	Add and store. This is an overloaded operator which mean that it's behavior depend on the given operands.
$x **-=** *expr*	x = x - y	Subtract and store.
$x ***=** *expr*	x = x * y	Multiply and store.
$x **/=** *expr*	x = x / y	Divide and store.
$x **%=** *expr*	x = x % y	Modulo and store.
$x **<<=** *expr*	x = x << y	Left shift and store.
$x **>>=** *expr*	x = x >> y	Right shift and store.
$x **.=** *expr*	x = x .. y	Concatenate string and store.
$x **&=** *expr*	x = x & y	Bit-and and store.
$x **^=** *expr*	x = x ^ y	Bit-xor and store.
$x **|=** *expr*	x = x | y	Bit-or and store.

Note that the assignment copies the original variable to the new one (assignment by value), so changes to one will not affect the other.

An exception to the usual assignment by value occurs when dealing with JSON arrays and objects which are assigned by reference (for performance reason). That is, when assigning an already created instance of a JSON array or object to a variable, the new variable will access the same instance as the array or object that was assigned. In other words, arrays and objects are shared between variables. This behavior is the same when passing arrays

or objects to a function.

Arithmetic Operators

Arithmetic operators take numerical (64 integers or real numbers) values (either literals, variables or even strings) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, 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.)

In addition, Jx9 provides the arithmetic operators listed in the following table.

Arithmetic Operators

Operator	Description	Example
++	Unary increment operator. Adds one to its operand. If used as a prefix operator (++$x), returns the value of its operand after adding one; if used as a postfix operator ($x++), returns the value of its operand before adding one.	If **$x** is 10, then **++$x**; sets **$x** to 11 and returns 11, whereas **$x++**; returns 10 and, only then, sets **$x** to 11.
--	Unary decrement operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If **$x** is 10, then **--$x**; sets **$x** to 9 and returns 9, whereas **$x--**; returns 10 and, only then, sets **$x** to 9.
%	Modulus: Returns the integer remainder of dividing the two operands.	print 1024 % 0xff; //4
-	Unary negation operator. Returns the negation of its operand.	$x = -15; print $x; //-15

Comparison Operators

Comparison operators, as their name implies, allow you to compare two values.

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared byte per byte using a memcmp() like function. In most cases, if the two operands are not of the same type, Jx9 attempts to convert the operands to an appropriate type for the comparison. (The sole exceptions to this rule are === and !==, which perform "strict" equality and inequality and which do not attempt to convert the operands to compatible types before checking equality.) The following table describes the comparison operators, assuming the following code:

$x = 6, $y = 10

Comparison operators

Operator	Description	Examples
== Equal	Returns true if the operands are equal.	6 == $x "6" == $x 10 == 0xA
!= Not equal	Returns true if the operands are not equal.	$x != 10

Operator	Description	Examples
		$y != "6"
<> Not equal	Returns true if the operands are not equal.	$x <> 10 $y <> "6"
=== Strict equal	Returns true if the operands are equal and of the same type.	6 === $x
!== Strict not equal	Returns true if the operands are not equal and/or not of the same type.	$x !== "6" 6 !== '6'
> Greater than	Returns true if the left operand is greater than the right operand.	$x > $y '12 sheep' > 0xA
>= Greater than or equal	Returns true if the left operand is greater than or equal to the right operand.	$x >= $y $x >= 100
< Less than	Returns true if the left operand is less than the right operand.	$x < $y
<= Less than or equal	Returns true if the left operand is less than or equal to the right operand.	$x <= $y $x <= 10

Ternary Operator

The ternary operator is the only Jx9 operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

condition ? expr1 : expr2 ;

If condition is true, the operator has the value of expr1. Otherwise it has the value of expr2. You can use the conditional operator anywhere you would use a standard operator.

Example:

```
$rank = 6;
$status = $rank > 5 ? "good" : "bad";
print $status; //good
```

Bitwise Operators

Bitwise operators treat their operands as a set of 64 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard Jx9 numerical values.

The following table summarizes Jx9 bitwise operators:

Operator	Usage	Description
Bit AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bit OR	a | b	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bit XOR	a ^ b	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bit NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b bits to the left, shifting in zeros from the right.

Operator	Usage	Description
Right shift	a >> b	Shifts a in binary representation b bits to the right, discarding bits shifted off.

Logical Operators.

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Table 3.6 Logical operators

Operator	Usage	Description
&&	expr1 && expr2	(Logical AND) Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
||	expr1 || expr2	(Logical OR) Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
!	!expr	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0 or the empty string ("").

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- false && *anything* is short-circuit evaluated to false.
- true || *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Example:

```
//foo() will never get called as those operators are short-circuit

$x = false && foo(); // $x = false;
$y = true || foo();   // $y = true;
```

String Operators.

In addition to the comparison operators, which can be used on string values, the concatenation operator **..** (*two dots*) concatenates two string values together, returning another string that is the union of the two operand strings. For example, "my " .. "string" returns the string "my string".

The shorthand assignment operator .= can also be used to concatenate strings.

Example:

```
  $date = 'Current date is: ' .. __DATE__;
```

```
print $date .. JX9_EOL; //Current date is: 2013-01-06
//Append a single space and the current time
$date .= ' ' .. __TIME__;
print $date  //Current date is: 2013-01-06 11:58:02
```

Member Access Operators.

Member operators provide access to an object's properties and arrays entries.

A JSON object is actually an *associative array* (i.e. *map*, *dictionary*, *hash*). The *keys* in this hash are the names of object members.

There are two ways to access array and object members: dot notation and bracket notation (i.e: subscript operator).

Dot notation

Here, simply separate the object name and the target member (including anonymous functions) with a dot as follows:

```
//declare a simple JSON object
$my_object = {
 'time' : __TIME__, //Current time
 'date' : __DATE__, //Current date
 'os' : uname() //Host operating system
};
//Output object members values
print $my_object.time; //14:27:32
print JX9_EOL; //\n
print $my_object.date; //2012-01-06
print JX9_EOL;
print $my_object.os; //Ubuntu Linux ...
```

Object that declare another object:

```
$person = {
   firstName : "John",
   lastName: "Smith",
   age: 25,
   addr: {
      "streetAddress": "21 2nd Street",
      city: "New York",
      state : "NY",
      postalCode : 10021
   }
}; //Don't forget the semi-colon here

print $person.addr.city; //New York
print JX9_EOL;
print $person.addr.postalCode; //10021
```

Bracket notation

Bracket notation or subscript operator is used to access JSON array fields where their keys are usually numeric and are automatically assigned by the underlying Jx9 virtual machine.

The notation is

$array_name[expr];

Example:

```
//Declare a JSON array with three fields
$my_array = [ __TIME__, __DATE__, uname() ];
print $my_array[0]; //14:24:05
print JX9_EOL;
print $my_array[1]; //2013-01-06
print JX9_EOL;
print $my_array[2]; //Windows 8...
```

The subscript operator is used also to populate JSON objects/arrays with new members (i.e. Key/Value pair) after their instantiation as follows:

```
//Declare an empty JSON object
$my_info = {};
//Populate the object
$my_info['name'] = 'John Smith';
$my_info['age'] = 27;
print $my_info; // { "name" : "John Smith", "age" : 27 }
```

JSON Array/Object Operators

Operators Table

Operator	Usage	Description
+	Union	Union of two JSON array or object.
==	Equality	**TRUE** if the two object have the same key/value pairs.
!=	Inequality	**TRUE** if *$y* is not equal to *$y*.

Example:

```
//Declare two JSON objects and perform their union.
$a = {
    a :  "apple",
    b : "banana"
 };
$b =  {
     a : "pear",
```

```
    b : "strawberry",
    c  : "cherry"
  };

$c = $a + $b; // Union of $a and $b
print "Union of \$a and \$b: \n";
print $c;

$c = $b + $a; // Union of $b and $a
print "Union of \$b and \$a: \n";
print $c;
```

When running, you should see something like that:

```
Union of $a and $b:
{ "a" : "apple", "b" : "banana", "c": "cherry" }
Union of $b and $a:
{ "a" : "pear", "b" : "strawberry", "c" : "cherry" }
```

The Jx9 standard library includes a set of useful functions for working with JSON arrays and objects (i.e. sorting, merging, diff functions and so forth). Refer to the built-in functions page for additional information.

Comma Operator.

A comma expression contains two operands of any type separated by a comma and has left-to-right associativity. The left operand is fully evaluated, possibly producing side effects, and its value, if there is one, is discarded. The right operand is then evaluated. The type and value of the result of a comma expression are those of its right operand, after the usual unary conversions.

Any number of expressions separated by commas can form a single expression because the comma operator is associative. The use of the comma operator guarantees that the sub-expressions will be evaluated in left-to-right order, and the value of the last becomes the value of the entire expression.

The following example assign the value 25 to the variable $a, multiply the value of $a with 2 and assign the result to variable $b and finally we call a test function to output the value of $a and $b. Keep-in mind that all theses operations are done in a single expression using the comma operator.

Example:

```
$a = 25 , $b = $a << 1 , test();
/* Output the value of $a and $b */
function test(){
 uplink $a,$b;
 print "\$a = $a \$b= $b\n"; /* You should see: $a = 25 $b = 50*/
}
```

The primary use of comma expressions is to produce side effects in the following situations:

- Calling a function
- Entering an iteration loop
- Testing a condition

In some contexts where the comma character is used, parentheses are required to avoid

ambiguity.

Operators Precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

The following table describes the precedence of operators, from highest to lowest.

Operator Name	Operator
Member Access	**. []**
Function Call	()
Increment/Decrement	**++ --**
Unary/Negation	**- + ! ~**
Type Cast	**(int) (string) (float) (bool)**
Multiply/Divide/Modulo	*** / %**
Addition/Subtraction/Concatenation	**+ - ..**
Bitwise shift	**<< >>**
Relational	**> >= < <=**
Equality	**<>**
Equality	**== != === !==**
Bitwise-and	**&**
Bitwsise-xor	**^**
Bitwise-or	**|**
Logical and	**&&**
Logical or	**||**
Ternary	**?:**
Assignment	**= += -= *= /= %= .= &= |= ^= <<= >>=**
Comma	**,**

Statements

a statement is the smallest standalone element of an imperative programming language. A program written in such a language is formed by a sequence of one or more statements. (src: Wikipedia)

Any expression is also a statement. See Expressions and Operators for additional information.

Use the semicolon **;** character to separate statements in Jx9 code.

Block Statements

A block statement is used to group statements. The block is delimited by a pair of curly

brackets:

```
{
 stmt_1;
 stmt_2;
 ...
 stmt_n;
}
```

Block statements are commonly used with control flow statements (i.e. if, for, while) as follows:

```
$x = 0;
while( $x++ < 5 ){
 print "$x\n";
}
```

Conditionals

A conditional statement is a set of commands that executes if a specified condition is true. Jx9 supports two conditional statements: **if...else** and **switch**.

if..else..else if..elseif

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false. You may also compound the statements using **else if** or **elseif** to have multiple conditions tested in sequence.

An if statement looks as follows:

```
if (condition)
 statement_1
[else if (condition_2)
 statement_2]
...
[elseif (condition_n_1)
 statement_n_1]
[else
 statement_n]
```

condition can be any expression that evaluates to true or false. If condition evaluates to true, statement_1 is executed; otherwise, statement_2 is executed. statement_1 and statement_2 can be any statement, including further nested if statements.

To execute multiple statements, use a block statement ({ ... }) to group those statements. In general, it is a good practice to always use block statements, especially in code involving nested if statements:

```
if (condition) {
 stmts_1
}else if (condition2){
 stmts_2
}else{
```

```
 stmts_3
}
```

Example:

```
$x = 10, $y = 20;
if( $x > $y ){
 print "\$x is greater than \$y\n";
}elseif ($x < $y ){
 print "\$x is smaller than \$y\n";
}else{
 print "\$x and \$y are equals\n";
}
```

While Statement

while loops are the simplest type of loop in Jx9. 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 Jx9 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 (each time Jx9 runs the statements in the loop is one iteration). Sometimes, if the *while* expression evaluates to **FALSE** from the very beginning, the nested statement(s) won't even be run once.

Example:

```
//Loop until $x >= 5
$x = 0;
while( $x++ < 5 ){
 print "$x\n";
}
```

For Statement

for statement are one of the most powerful looping mechanism in Jx9. They behave like their C counterparts. The syntax of a *for* loop is:

```
for (init_expr; cond_expr; post_expr)
    statement
```

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

In the beginning of each iteration, *cond_expr* 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, *post_expr* is evaluated (executed).

Each of the expressions can be empty or contain multiple expressions separated by commas. In *cond_expr*, all expressions separated by a comma are evaluated but the result is taken from the last part. *cond_expr* being empty means the loop should be run indefinitely (Jx9 implicitly considers it as **TRUE**, like C). This may not be as useless as you might think, since often you'd want to end the loop using a conditional *break* statement instead of using the *for* truth expression.

Example:

```
/* example 1 */

for ($i = 1; $i <= 10; $i++) {
   print $i;
}

/* example 2 */

for ($i = 1; ; $i++) {
   if ($i > 10) {
      break;
   }
   print $i;
}

/* example 3 */

$i = 1;
for (; ; ) {
   if ($i > 10) {
      break;
   }
   print $i;
   $i++;
}

/* example 4 */

for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
```

---

break/continue statements

break ends execution of the current *for*, *foreach*, *while* or *switch* structure.

break accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of.

Example:

```
/* Using the optional argument. */
$i = 0;
while (++$i) {
```

```
    switch ($i) {
    case 5:
        print "At 5\n";
        break 1;  /* Exit only the switch. */
    case 10:
        print "At 10; quitting\n";
        break 2;  /* Exit the switch and the while. */
    default:
        break;
    }
}
```

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.

continue accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of.

> **Note**: *continue 0;* and *continue 1;* is the same as running *continue;*.

Example:

```
$i = 0;
while ($i++ < 5) {
    print "Outer\n";
    while (1) {
        print "Middle\n";
        while (1) {
            print "Inner\n";
            continue 3;
        }
        print "This never gets output.\n";
    }
    print "Neither does this.\n";
}
```

---

foreach statement

The *foreach* construct simply gives an easy way to iterate over JSON arrays or objects. *foreach* works only on arrays and objects, and will issue a run-time warning when you try to use it on a variable with a different data type or an uninitialized variable. There are two syntaxes; the second is a minor but useful extension of the first:

```
foreach (array_or_object_expr as $value)
    statement;
foreach ( array_or_object_expr as $key , $value)
    statement;
```

The first form loops over the target array or object given by *array_or_object_expr*. On each loop, the value of the current element is assigned to *$value* and the internal array/object cursor is advanced by one (so on the next loop, you'll be looking at the next element).

The second form does the same thing, except that the current element's key will be

assigned to the variable *$key* on each loop.

***Note**: When foreach first starts executing, the internal array/object pointer is automatically reset to the first element of the array.*

Example:

```
/* Iterate over a JSON object */
$person = {
   firstName : "John",
   lastName: "Smith",
   age: 25,
   addr: {
      "streetAddress": "21 2nd Street",
      city: "New York",
      state : "NY",
      postalCode : 10021
   }
}; //Don't forget the semi-colon here

foreach( $person as $key, $value ){
  print "$key ==> $value\n";
}
```

Example2:

```
/* Iterate over a JSON array */
foreach( [5,6,7,8,9,10] as $value ){
  print "$value\n";
 }
```

---

switch statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

A switch statement looks as follows:

```
switch(expr){
 case cond1:
   stmt1;
   [break;]
 case cond2:
  stmt2;
  [break;]
 [default:
   def_stmt;
   [break;]
 ]
```

}

The program first looks for a case clause with a label matching the value of expression (Any complex expression including function calls) and then transfers control to that clause, executing the associated statements. If no matching label is found, the program looks for the optional default clause, and if found, transfers control to that clause, executing the associated statements. If no default clause is found, the program continues execution at the statement following the end of switch. By convention, the default clause is the last clause, but it does not need to be so.

The optional break statement associated with each case clause ensures that the program breaks out of switch once the matched statement is executed and continues execution at the statement following switch. If break is omitted, the program continues execution at the next statement in the switch statement.

> **Note**: Unlike some other languages, the continue statement applies to switch and acts similar to *break*. If you have a switch inside a loop and wish to continue to the next iteration of the outer loop, use *continue 2*.

Example:

```
$i = 'banana';

switch ($i) {
   case "apple":
      print "i is apple";
      break;
   case "banana":
      print "i is banana";
      break;
   case "cake":
      print "i is cake";
      break;
}
```

---

return statement

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.

If called from the global scope, then execution of the current script file is ended. If the current script file was include()ed or import()ed, then control is passed back to the calling file. Furthermore, if the current script file was include()ed, then the value given to **return()** will be returned as the value of the include() call.

> **Note**: If no parameter is supplied, then NULL is returned to the caller.

---

include

The **include** statement includes and evaluates the specified file.

Files are included based on the file path given or, if none is given, **include** will finally check in the calling script's own directory and the current working directory before failing. The **include** construct will emit a run-time **warning** if it cannot find a file.

If a path is defined — whether absolute (starting with a drive letter or \ on Windows, or / on Unix/Linux systems) or relative to the current directory (starting with . or ..) — the registered paths will be ignored altogether. For example, if a filename begins with ../, the Jx9 Virtual Machine will look in the parent directory to find the requested file.

When a file is included, the code it contains inherits the variable scope of the line on which the include occurs. Any variables available at that line in the calling file will be available within the called file, from that point forward. However, all functions and classes defined in the included file have the global scope.

If the include occurs inside a function within the calling file, then all of the code contained in the called file will behave as though it had been defined inside that function.

Example of includes:

```
include 'driver.jx9'; //Local relative path

include '/usr/local/lib/main.jx9'; //Local absolute path

include 'http://jx9.symisc.net/driver.jx9'; //Remote include
```

> **Note**: The remote include facility will be available only if the HTTP IO stream is registered within the target Jx9 Virtual Machine (which is not the case in the default build).

---

import

The **import** statement includes and evaluates the specified file during the execution of the script. This is a behavior similar to the include() statement, with the only difference being that if the code from a file has already been included, it will not be included again. That is, it will be included just once.

import may be used in cases where the same file might be included and evaluated more than once during a particular execution of a script, so in this case it may help avoid problems such as function redefinitions, variable value reassignments, etc.

See the include documentation for information about how this construct works.

Language Constructs

The Jx9 language constructs resemble to a standard function except that you do not need parenthesis to invoke them. These constructs are:

print

print expr*[, expr];*

The print construct is used to output one or more expressions to the default VM output consumer callback.

die

die *[expr]*;

This construct when invoked output a message if available and terminate the current script.

Example:

```
die;
```

```
die 'Giving Up!!';
```

Function

Functions are one of the fundamental building blocks in Jx9. A function is a Jx9 procedure —a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

A **function definition** (also called a **function declaration**) consists of the function keyword, followed by

- The name of the function.
- A list of arguments to the function with optional type, enclosed in parentheses and separated by commas.
- The Jx9 statements that define the function, enclosed in curly braces, { }.

For example, the following code defines a simple function named square:

```
function square($number)
{
 return $number * $number;
}
```

The same function could be declared using the type hinting feature as follows:

```
function square(int $number)
{
 return $number * $number;
}
```

With type hinting, arguments are automatically and silently converted to the desired type, here the integer type in our example.

Scalar parameters values (integer, string, floats and boolean) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function.

If you pass a JSON object or a JSON array, and the function changes the object's properties, that change is visible outside the function. This is called **pass by reference**.

A function can be recursive; that is, it can call itself. For example, here is a function that computes factorials recursively:

```
function factorial(int $num){
  if(  $num == 0 || $num == 1 )
    return 1;
   else
     return $num * factorial($num – 1);
}
print factorial(5); //120
```

> ***Note:*** The recursion limit (i.e. Total number of times a function may call itself) can be controlled by the host application via the jx9_vm_config() interface with a configuration verb set to **JX9_VM_CONFIG_RECURSION_DEPTH**.
>
> ***Note:*** *Function names under Jx9 are case-sensitive which mean that* ***Foo()*** *and*

foo() *are completely different.*

Jx9 supports the concept of variable functions. This means that if a variable name has parentheses appended to it, Jx9 will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it. Among other things, this can be used to implement callbacks, function tables, and so forth.

Example

```
function hello(string $name)
{
  print "Hello $name\n";
}
//Assign the function name to the variable $x
$x = 'he'..'llo';
//Invoke whatever $x evaluate to
$x('Dean'); //Hello Dean
```

Function Arguments

Information may be passed to functions via the argument list, which is a comma-delimited list of expressions.

A function may define C++-style default values for its arguments which can be any complex expression including function call.

Example:

```
function test(string $name = 'user_id_'..rand_str(4), int $age = 10 * 2 + 5)
{
  print "Name = $name\n";
  print "Age = $age\n";
}
/* Call without arguments */
test(); /* You should see: name = user_id_resr age = 25 */
/* Call with a single argument */
test('Me'); /* You should see: name = Me age = 25 */
```

Jx9 support variable-length argument lists in user-defined functions. This is really quite easy, using the func_num_args(), func_get_arg(), and func_get_args() functions.

No special syntax is required, and argument lists may still be explicitly provided with function definitions and will behave as normal.

Returning values

Refer to the return statement.

Anonymous Function

Anonymous functions, also known as *lambda*, allow the creation of functions which have no specified name. They are most useful as the value of callback parameters, but they have many other uses.

Anonymous functions are declared exactly like a standard function but without the associated identifier (i.e. The function name).

Example:

```
$greet = function($name)
{
   printf("Hello %s\n", $name);
};
$greet('World'); //Hello World
```

Like any other expressions, anonymous function can be passed as an argument to a function as follows:

```
function main($callback)
{
  if( !is_callable($callback) )
     print "Not callable\n";
   else
     //Invoke the given callback
     $callback();
 }
//Pass a simple callback that display a greeting message
main(function(){ print "Hello World\n"; }); //Hello world
```

Function overloading

Function overloading *is a feature found in various programming languages such as C++ or Java, that allows creating several function and/or methods with the same name which differ from each other in the type of the input and the output of the function. It is simply defined as the ability of one function to perform different tasks. (src: Wikipedia)*

Function overloading means two or more functions can have the same name but either the number of arguments or the data type of arguments has to be different.

Jx9 has support for this powerful mechanism. That is, you define two ore more standard Jx9 functions with the same name but with different arguments number and/or signature and finally you perform a simple function call and let Jx9 peek the appropriate function for this context.

Example:

```
// volume of a cube
function volume(int $s)
{
  return $s*$s*$s;
}
```

```
// volume of a cylinder
function volume(float $r,int $h)
{
  return 3.14*$r*$r*$h;
}
// volume of a cuboid
function volume(float $l,int $b,int $h)
{
  return $l*$b*$h;
}
/* Test the overloading mechanism */
print volume(10)..JX8_EOL; /* 1000 */
print volume(2.5,8)..JX9_EOL; /* 157 */
print volume(100,75,15); /* 112500 */
```