LK Scripting Language Reference

Aron P. Dobos

January 30, 2013

Abstract

The LK (Language Kit) language is a simple but powerful scripting language that is designed to be small, fast, and easily embedded in other applications. It allows users to extend the built-in functionality of programs, and provides a cross-platform standard library of function calls. The core LK engine, including lexical analyzer, parser, and evaluator comprises less than 4000 lines of ISO-standard C++ code, and is only dependent on the Standard C++ Library (STL), making it extremely tight and portable to various platforms. LK also provides a C language API for writing extensions that can be dynamically loaded at runtime.

Note: LK can utilize standard 8-bit ASCII strings via the built-in `std::string` class, or can be configured to utilize an `std::string`-compliant string class from an external library. In this way, Unicode text can be supported natively. For example, direct integration and linkage with the wxWidgets C++ GUI library string class is provided as an option.

Contents

1 Introduction

1.1 Hello world!

As with any new language, the traditional first program is to print the words "Hello, world!" on the screen, and the LK version is listed below.

```
out( "Hello, world!\n" );
```

Notable features:

1. The `out` command generates text output from the script
2. The text to be printed is enclosed in double-quotes
3. To move to a new line in the output window, the symbol `\n` is used
4. The program statement is terminated by a semicolon `;`

To run Hello world, type it into a new text file with the .lk extension, and load the script into the appropriate the LK script input form in your application.

1.2 Why yet another scripting language?

There are many scripting languages available that have all of LK's features, and many more. However, LK is unique in its code footprint (less than 4000 lines of C++), portability, and simplicity of integration into other programs. It is also very straightforward to write libraries of functions called extensions that can be dynamically loaded into the script engine at run-time.

Some notable aspects of LK include:

1. Functions are first-class values, so that they can be passed to procedures and defined implicitly
2. No distinction between functions and procedures, and they can be nested and mutually recursive
3. Built-in support for string indexed tables (hashes)
4. Initializer lists for arrays and tables
5. Fully automatic memory management and highly optimized internal data representation
6. Simple syntactic sugar to represent structures
7. '`elseif`' statement formatting is similar to PHP
8. '`for`' loop syntax follows the C / Perl convention

2 Data Variables

2.1 General Syntax

In LK, a program statement is generally placed on a line by itself, and a semicolon `;` marks the end of the statement. A very long program statement can be split across multiple lines to improve readability, provided that the line breaks do not occur in the middle of a word or number.

Blank lines may be inserted between statements. While they have no meaning, they can help make a script easier to read. Spaces can also be added or removed nearly anywhere, except of course in the middle of a word. The following statements all have the same meaning.

```
out("Hello 1\n");
  out  (
    "Hello 1\n");
out (    "Hello 1\n"  );
```

Comments are lines in the program code that are ignored by LK. They serve as a form of documentation, and can help other people (and you!) more easily understand what the script does. There a two types of comments. The first type begins with two forward slashes `//` , and continues to the end of the line.

```
// this program creates a greeting
out( "Hello, world!\n" ) // display the greeting to the user
```

The second type of comment is usually used when a large section of code needs to be ignored, or a paragraph of documentation is included with the program text. In this case, the comment begins with `/*` and ends with `*/` , as in the example below.

```
/* this program creates a greeting
   there are many interesting things to note:
    - uses the 'out' command
    - hello has five characters
    - a new line is inserted with an escape sequence
  */
out( "Hello, world!\n" ); // display the greeting to the user
```

2.2 Variables

Variables store information while your script is running. LK variables share many characteristics with other computer languages.

1. Variables do not need to be "declared" in advance of being used
2. There is no distinction between variables that store text and variables that store numbers
3. In LK, a variable can also be an array, table, or function

Variable names may contain letters, digit, and the underscore symbol. A limitation is that variables cannot start with a digit. Like some languages like C and Perl, LK does distinguish between upper

and lower case letters in a variable (or subroutine) name. As a result, the name `myData` is different from `MYdata`.

Values are assigned to variables using the equal sign `=`. Some examples are below.

```
Num_Modules = 10;
ArrayPowerWatts = 4k;
Tilt = 18.2;
system_name = "Super PV System";
Cost = "unknown";
COST = 1e6;
cost = 1M;
```

Assigning to a variable overwrites its previous value. As shown above, decimal numbers can be written using scientific notation or engineering suffixes. The last two assignments to `Cost` are the same value. Recognized suffixes are listed in the table below. Suffixes are case-sensitive, so that LK can distinguish between `m` (milli) and `M` (Mega).

Name	Suffix	Multiplier
Tera	T	1e12
Giga	G	1e9
Mega	M	1e6
Kilo	k	1e3
Milli	m	1e-3
Micro	u	1e-6
Nano	n	1e-9
Pico	p	1e-12
Femto	f	1e-15
Atto	a	1e-18

Table 1: Recognized Numerical Suffixes

2.3 Arithmetic

LK supports the five basic operations `+`, `-`, `*`, `/`, and `^`. The usual algebraic precendence rules are followed, so that multiplications and divisions are performed before additions and subtractions. The exponentiation operator `^` is performed before multiplications and divisions. Parentheses are also understood and can be used to change the default order of operations. Operators are left-associative, meaning that the expression `3-10-8` is understood as `(3-10)-8` .

More complicated operations like modulus arithmetic are possible using built-in function calls in the standard LK library.

Examples of arithmetic operations:

```
battery_cost = cost_per_kwh * battery_capacity;

// multiplication takes precedence
```

```
degraded_output = degraded_output - degraded_output * 0.1;

// use parentheses to subtract before multiplication
cash_amount = total_cost * ( 1 - debt_fraction/100.0 );
```

2.4 Simple Input and Output

You can use the built-in `out` and `outln` functions to write textual data to the console window. The difference is that `outln` automatically appends a newline character to the output. To output multiple text strings or variables, use the `+` operator, or separate them with a comma.

```
array_power = 4.3k;
array_eff = 0.11;
outln("Array power is " + array_power + " Watts.");
outln("It is " + (array_eff*100) + " percent efficient.");
outln("It is ", array_eff*100, " percent efficient."); // same as above
```

The console output generated is:

```
Array power is 4300 Watts.
It is 11 percent efficient.
```

Use the `in` function to read input from the user. You can optionally pass a message to `in` to display to the user when the input popup appears. The `in` function always returns a string, and you may need to convert to a different type to perform mathematical operations on the result.

```
cost_per_watt = to_real(in("Enter cost per watt:"));  // Show a message.  in() also is fine.
notice( "Total cost is: " + cost_per_watt * 4k + " dollars"); // 4kW system
```

The `notice` function works like `out`, except that it displays a pop up message box on the computer screen.

2.5 Data Types and Conversion

LK supports two basic types of data: numbers and text. Numbers can be integers or real numbers. Text strings are stored as a sequence of individual characters, and there is no specific limit on the length of a string. LK does not try to automatically convert between data types, and will issue an error if you try to multiply a number by a string, even if the string contains the textual representation of a valid number. To convert between data types, LK has several functions in the standard library for this purpose.

Boolean (true/false) data is also stored as a number - there is no separate boolean data type. All non-zero numbers are interpreted as true, while zero is false. This convention follows the C programming language.

There is also a special data value used in LK to indicate the absence of any value, known as the `null` value. It useful when working with arrays and tables of variables, which will be discussed later

in this document. When a variable's value is `null`, it cannot be multiplied, added, or otherwise used in a calculation.

Data Type	Conversion Function	Valid Values
Integer Number	`to_int()`	approx. +/- 1e-308 to 1e308
Real Number	`to_real()`	+/- 1e-308 to 1e308, not an number (NaN)
Boolean	`to_bool()`	`"true"` or `"false"` (`1` or `0`)
Text Strings	`to_string()`	Any length text string

Table 2: Intrinsic Data Types

Sometimes you have two numbers in text strings that you would like to multiply. This can happen if you read data in from a text file, for example. Since it does not make sense to try to multiply text strings, you need to first convert the strings to numbers. To convert a variable to a double-precision decimal number, use the `to_real` function, as below.

```
a = "3.5";
b = "-2";
c1 = a*b; // this will cause an error when you click 'Run'
c2 = to_real(a) * to_real(b); // this will assign c2 the number value of -7
```

The assignment to `c1` above will cause the error *error: access violation to non-numeric data*, while the assignment to `c2` makes sense and executes correctly.

You can also use `to_int` to convert a string to an integer or truncate a decimal number, or the `to_string` function to explicitly convert a number to a string variable.

If you need to find out what type a variable currently has, use the `typeof` function to get a text description.

```
a = 3.5;
b = -2;
c1 = a+b; // this will set c1 to -1.5
c2 = to_string( to_int(a) ) + to_int( b ); // c2 set to text "3-2"

outln( typeof(a) ); // will display "number"
outln( typeof(c2) ); // will display "string"
```

2.6 Special Characters

Text data can contain special characters to denote tabs, line endings, and other useful elements that are not part of the normal alphabet. These are inserted into quoted text strings with *escape sequences*, which begin with the `\` character.

So, to print the text `"Hi, tabbed world!"`, or assign `c:\Windows\notepad.exe`, you would have to write:

```
outln("\"Hi,\ttabbed world!\"")
program = "c:\\Windows\\notepad.exe"
```

Escape Sequence	Meaning
`\n`	New line
`\t`	Tab character
`\r`	Carriage return
`\"`	Double quote
`\\`	Backslash character

Table 3: Text String Escape Sequences

Note that for file names on a Windows computer, it is important to convert back slashes (`'\'`) to forward slashes (`/`). Otherwise, the file name may be translated incorrectly and the file won't be found.

2.7 Constants and Enumerations

LK allows constants to be specified using the `const` keyword. Any type of variable can be declared constant, including numbers, strings, and functions (described later). Attempting to assign a new value to a `const` variable will result in a run-time error when the script is executed. Also, once a variable has been declared, it cannot be re-declared as `const` in a subsequent part of the program. Arrays and tables behave slightly differently from numbers: while the variable name referring to the array or table cannot be changed to another data type, the actual array or table contents are not protected by the `const` specifier. Examples:

```
const pi = 3.1415926;
pi = 3.11;       // will cause an error

value = 43;
const value = 51;     // will cause an error (value already exists)


const names = [ "Linus", "David", "Aron" ];
names[2] = "Ari";    // allowed because it changes the contents of array 'names'
                     // but not the data type or size of 'names'

names[3] = "Nora";   // not allowed because it changes the size of the 'names' array
names = "Patrick";    // will cause an error


const sqr = define(x) { return x*x; };  // function sqr can't being changed
```

Sometimes it is desirable to specify many constants, perhaps to define various states in a program. For this purpose, an `enum`, or *enumerate* statement exists. The state of a motor could be indicated with the constants below, instead of with numbers, which makes a program much more readable.

```
enum { STARTING, RUNNING, STOPPED, ERROR  };

motor_state = STOPPED;
```

Enumerations start assigning integer values to the names in the list, increasing by one. It is possible to assign a custom values to names in the following ways:

```
enum { STARTING, RUNNING, STOPPED, EFF=+20, EFF1, EFF2,  ERROR1 = 100, ERROR2 };
```

In this case, everything is the same as before until the EFF variable, for which the `+20` specifies that it should have a value 20 greater than the previous name in the enumeration: 22. EFF1 and EFF2 have values 23 and 24. It is also possible to jump to a known value, as with ERROR1.

The `enum` statement is simply a more convenient syntax to make to multiple `const` assignments. The enumeration above is semantically equivalent to:

```
const local STARTING = 0;
const local RUNNING = STARTING + 1;
const local STOPPED = RUNNING + 1;
const local EFF = STOPPED + 20;
const local EFF1 = EFF + 1;
const local EFF2 = EFF1 + 1;
const local ERROR1 = 100;
const local ERROR2 = ERROR1 + 1;
```

The meaning of the `local` specifier will be explained in later sections.

3 Flow Control

3.1 Comparison Operators

LK supports many ways of comparing data. These types of tests can control the program flow with branching and looping constructs that we will discuss later.

There are six standard comparison operators that can be used on most types of data. For text strings, "less than" and "greater than" are with respect to alphabetical order.

Comparison	Operator
Equal	`==`
Not Equal	`!=`
Less Than	`<`
Less Than or Equal	`<=`
Greater Than	`>`
Greater Than or Equal	`>=`

Table 4: Comparison Operators

Examples of comparisons:

```
divisor != 0
state == "oregon"
error <= -0.003
"pv" > "csp"
```

Single comparisons can be combined by *boolean* operators into more complicated tests.

1. The `!` operator yields true when the test is false. It is placed before the test whose result is to be notted.
   Example: `!(divisor == 0)`
2. The `&&` operator yields true only if both tests are true.
   Example: `divisor != 0 && dividend > 1`
3. The `||` operator yields true if either test is true.
   Example: `state == "oregon" || state == "colorado"`

The boolean operators can be combined to make even more complex tests. The operators are listed above in order of highest precedence to lowest. If you are unsure of which test will be evaluated first, use parentheses to group tests. Note that the following statements have very different meanings.

```
state_count > 0 && state_abbrev == "CA" || state_abbrev == "OR"
state_count > 0 && (state_abbrev == "CA" || state_abbrev == "OR")
```

3.2 Branching

Using the comparison and boolean operators to define tests, you can control whether a section of code in your script will be executed or not. Therefore, the script can make decisions depending on different circumstances and user inputs.

3.2.1 `if` Statements

The simplest branching construct is the `if` statement. For example:

```
if ( tilt < 0.0 )
{
    outln("Error: tilt angle must be 0 or greater")
}
```

Note the following characteristics of the `if` statement:

1. The test is placed in parentheses after the `if` keyword.
2. A curly brace {indicates a new block of code statements.
3. The following program lines include the statements to execute when the `if` test succeeds.
4. To help program readability, the statements inside the `if` are usually indented.
5. The construct concludes with the } curly brace to indicate the end of the block..

6. When the `if` test fails, the program statements inside the block are skipped.

3.2.2 `else` Construct

When you also have commands you wish to execute when the `if` test fails, use the `else` clause. For example:

```
if ( power > 0 )
{
    energy = power * time;
    operating_cost = energy * energy_cost;
}
else
{
    outln("Error, no power was generated.");
    energy = -1;
    operating_cost = -1;
}
```

3.2.3 Multiple `if` Tests

Sometimes you wish to test many conditions in a sequence, and take appropriate action depending on which test is successful. In this situation, use the `elseif` clause. Be careful to spell it as a single word, as both `else if` and `elseif` can be syntactically correct, but have different meanings.

```
if ( angle >= 0 && angle < 90)
{
    text = "first quadrant";
}
elseif ( angle >= 90 && angle < 180 )
{
    text = "second quadrant";
}
elseif ( angle >= 180 && angle < 270 )
{
    text = "third quadrant";
}
else
{
    text = "fourth quadrant";
}
```

You do not need to end a sequence of `elseif` statements with the `else` clause, although in most cases it is appropriate so that every situation can be handled. You can also nest `if` constructs if needed. Again, we recommend indenting each "level" of nesting to improve your script's readability. For example:

```
if (   angle >= 0
    && angle < 90 ) {

    if ( print_value == true ) {
        outln( "first quadrant: " + angle );
    }  else  {
        outln( "first quadrant" );
    }
}
```

Also note that because LK does not care about spaces and tabs when reading the program text, it is possible to use multiple lines for a long if statement test to make it more readable. The curly braces denoting the code block can also follow on the same line as the `if` or on the next line.

3.2.4 Single statement `if`s

Sometimes you only want to take a single action when an `if` statement succeeds. To reduce the amount of code you must type, LK accepts single line `if` statements that do not include the { and } block delimiters, as shown below.

```
if ( azimuth < 0 ) outln( "Warning: azimuth < 0, continuing..." );

if ( tilt > 90 ) tilt = 90;   // set maximum tilt value
```

You can also use a `elseif` and/or `else` statement on single line `if`. Like the `if`, it only accepts one program statement, and must be typed on the same program line. Example:

```
if ( value > average ) outln("Above average");
  else outln("Not above average");
```

3.3 Looping

A loop is a way of repeating the same commands over and over. You may need to process each line of a file in the same way, or sort a list of names. To achieve such tasks, LK provides two types of loop constructs, the `while` and `for` loops.

Like `if` statements, loops contain a "body" of program statements followed by a closing curly bracke } to denote where the loop construct ends.

3.3.1 `while` Loops

The `while` loop is the simplest loop. It repeats one or more program statements as long as a logical test holds true. When the test fails, the loop ends, and the program continues execution of the statements following the loop construct. For example:

```
while ( done == false )
{
```

```
    // process some data
    // check if we are finished and update the 'done' variable
}
```

The test in a `while` loop is checked before the body of the loop is entered for the first time. In the example above, we must set the variable `done` to `false` before the loop, because otherwise no data processing would occur. After each iteration ends, the test is checked again to determine whether to continue the loop or not.

3.3.2 Counter-driven Loops

Counter-driven loops are useful when you want to run a sequence of commands for a certain number of times. As an example, you may wish to display only the first 10 lines in a text file.

There are four basic parts of implementing a counter-driven loop:

1. Initialize a counter variable before the loop begins.
2. Test to see if the counter variable has reached a set maximum value.
3. Execute the program statements in the loop, if the counter has not reached the maximum value.
4. Increment the counter by some value.

For example, we can implement a counter-driven loop using the `while` construct:

```
i = 0;      // use i as counter variable
while (i < 10) {
    outln( "value of i is " + i );
    i = i + 1;
}
```

3.3.3 `for` Loops

The `for` loop provides a streamlined way to write a counter-driven loop. It combines the counter initialization, test, and increment statements into a single line. The script below produces exactly the same effect as the `while` loop example above.

```
for ( i = 0; i < 10; i++ ) {
    outln( "value of i is " + i );
}
```

The three loop control statements are separated by semicolons in the `for` loop statement. The initialization statement (first) is run only once before the loop starts. The test statement (second) is run before entering an iteration of the loop body. Finally, the increment statement is run after each completed iteration, and before the test is rechecked. Note that you can use any assignment or calculation in the increment statement.

Note that the increment operator `++` is used to modify the counter variable i. It is equally valid to write `i=i+1` for the counter advancement expression. The `++` is simply shorthand for adding 1 to a number. The `--` operator similarly decrement a number by 1.

Just like the `if` statement, LK allows `for` loops that contain only one program statement in the body to be written on one line without curly braces. For example:

```
for ( val=57; val > 1; val = val / 2 ) outln("Value is " + val );
```

3.3.4 Loop Control Statements

In some cases you may want to end a loop prematurely. Suppose under normal conditions, you would iterate 10 times, but because of some rare circumstance, you must break the loop's normal path of execution after the third iteration. To do this, use the `break` statement.

```
value = to_real( in("Enter a starting value") );
for ( i=0; i<10; i=i+1 )
{
    if (value < 0.01)
    {
         break;
}
    outln("Value is " + value );
    value = value / 3.0;
}
```

In another situation, you may not want to altogether break the loop, but skip the rest of program statements left in the current iteration. For example, you may be processing a list of files, but each one is only processed if it starts with a specific line. The `continue` keyword provides this functionality.

```
for ( i=0; i<file_count; i++ )
{
    file_header_ok = false;

    // check if whether current file has the correct header

    if (!file_header_ok) continue;

    // process this file
}
```

The `break` and `continue` statements can be used with both `for` and `while` loops. If you have nested loops, the statements will act in relation to the nearest loop structure. In other words, a `break` statement in the body of the inner-most loop will only break the execution of the inner-most loop.

3.4 Quitting

LK script execution normally ends when there are no more statements to run at the end of the script. However, sometimes you may need to halt early, if the user chooses not to continue an operation.

The `exit` statement will end the script immediately. For example:

```
if ( yesno("Do you want to quit?") ) {
    outln("Aborted.");
    exit;
}
```

The `yesno` function call displays a message box on the user's screen with yes and no buttons, showing the given message. It returns `true` if the user clicked yes, or `false` otherwise.

4 Arrays

Often you need to store a list of related values. For example, you may need to refer to the price of energy in different years. Or you might have a list of state names and capital cities. In LK, you can use arrays to store these types of collections of data.

4.1 Initializing and Indexing

An *array* is simply a variable that has many values, and each value is indexed by a number. Each variable in the array is called an *element* of the array, and the position of the element within the array is called the element's *index*. The index of the first element in an array is always 0.

To access array elements, enclose the index number in square brackets immediately following the variable name. You do not need to declare or allocate space for the array data in advance. However, if you refer to an element at a high index number first, all of the elements up to that index are reserved and given the `null` value.

```
names[0] = "Sean";
names[1] = "Walter";
names[2] = "Pam";
names[3] = "Claire";
names[4] = "Patrick";

outln( names[3] ); //  output is "Claire"
my_index = 2;
outln( names[my_index] ); // output is "Pam"
```

You can also define an array using the `[ ]` initializer syntax. Simply separate each element with a comma. There is no limit to the number of elements you can list in an array initializer list.

```
names = ["Sean", "Walter", "Pam", "Claire", "Patrick"];
outln( "First: " + names[0] );
outln( "All: " + names );
```

Note that calling the `typeof` function on an array variable will return "array" as the type description, not the type of the elements. This is because LK is not strict about the types of variables stored in an array, and does not require all elements to be of the same type.

4.2 Array Length

Sometimes you do not know in advance how many elements are in an array. This can happen if you are reading a list of numbers from a text file, storing each as an element in an array. After the all the data has been read, you can use the `#` operator to determine how many elements the array contains.

```
count = #names;
```

4.3 Processing Arrays

Arrays and loops naturally go together, since frequently you may want to perform the same operation on each element of an array. For example, you may want to find the total sum of an array of numbers.

```
numbers = [ 1, -3, 2.4, 9, 7, 22, -2.1, 5.8 ];

sum = 0;
for (i=0; i < #numbers; i++)
    sum = sum + numbers[i];
```

The important feature of this code is that it will work regardless of how many elements are in the array `numbers`.

4.4 Multidimensional Arrays

As previously noted, LK is not strict with the types of elements stored in an array. Therefore, a single array element can even be another array. This allows you to define matrices with both row and column indexes, and also three (or greater) dimensional arrays.

To create a multi-dimensional array, simply separate the indices with commas between the square brackets. For example:

```
data[0][0] = 3
data[0][1] = -2
data[1][0] = 5
data[2][0] = 1
```

```
nrows = #data; // result is 4
ncols = #data[0] // result is 2

row1 = data[0]; // extract the first row

x = row1[0]; // value is 3
y = row1[1]; // value is -2
```

The array initializer syntax `[ ]` can also be used to declare arrays in multiple dimensions. This is often useful when declaring a matrix of numbers, or a list of states and associated capitals, for example.

```
matrix = [ [2,3,4],
           [4,5,6],
           [4,2,1] ];

vector = [ 2, 4, 5 ];

outln( matrix[0] ); // prints the first row [2,3,4]

list = [ ["oregon", "salem"],
         ["colorado", "denver"],
         ["new york", "albany"] ];
```

4.5 Managing Array Storage

When you define an array, LK automatically allocates sufficient computer memory to store the elements. If you know in advance that your array will contain 100 elements, for example, it can be much faster to allocate the computer memory before filling the array with data. Use the `alloc` command to make space for 1 or 2 dimensional arrays. The array elements are filled in with the `null` value.

```
data = alloc(3,2);  // a matrix with 3 rows and 2 columns
data[2][1] = 3;

prices = alloc( 5 ); // a simple 5 element array
```

As before, you can extend the array simply by using higher indexes.

5 Tables

In addition to arrays, LK provides another built-in data storage structure called a table. A table is useful when storing data associated with a specific key. Similar to how arrays are indexed by numbers, tables are indexed by text strings. The value stored with a key can be a number, text string, array, or even another table. A good example when it would be appropriate to use a

table is to store a dictionary in memory, where each word is a key that has associated with it the definition of the word. In other languages, tables are sometimes called dictionaries, or hashes or hash tables.

5.1 Initializing and Indexing

To refer to data elements in a table, enclose the text key string in curly braces immediately following the variable name. You do not need to declare or allocate space for the table.

```
wordlen{"name"} = 4;
wordlen{"dog"} = 3;
wordlen{"simple"} = 5;
wordlen{"lk"} = 2;

outln(wordlen); // prints '{ name=4 dog=3 simple=5 lk=2 }'
outln( wordlen{"simple"} ); // prints 5
outln( typeof( wordlen{"can"} ) ); // prints 'null' (the key is not in the table)

key = "dog";
num_letters = wordlen{key}; // num_letters is 3
```

Calling the `typeof` function on a table variable will return "table" as the type description. LK is not strict about the types of variables stored in a table, and does not require all of the elements to be of the same type.

You can also define a table with key=value pairs using the { } initializer syntax. The pairs are separated by commas, as shown below.

```
wordlen = { "antelope"=4, "dog"=3, "cat"=5, "frog"=2 };
outln( wordlen ); // prints { dog=3 frog=2 cat=5 antelope=4 }
```

Note the order of the printout of the key-value pairs in the example above. Unlike an array whose elements are stored sequentially in memory, tables store data in an unordered set. As a result, once a key=value pair is added to a table, there is no specification of order of the pair relative to the other key=value pairs already in the table.

5.2 Table Processing

It is often necessary to know how many key=value pairs are in a table, and to know what all the keys are. In many situations, you may have to perform an operation on every key=value pair in a table, but you do not know in advance what the keys are or how many there are. LK provides two operators # and @ for this purpose.

To remove a key=value pair from a table, simply assign the special `null` value to a key.

```
wordlen = { "antelope"=4, "dog"=3, "cat"=5, "frog"=2 };
outln("number of key,value pairs =" + #wordlen);
```

```
keys = @wordlen;
outln(keys);

wordlen{"frog"} = null; // erase the frog entry

keys = @wordlen;
outln(keys);
for (i=0;i<#keys;i++)
   outln("key '" + keys[i] + "'=" + wordlen{keys[i]});
```

5.3 Record Structures

Many languages provide the ability to create user-defined data types that are built from the intrinsic types. For example, an address book application might have a record for each person, and each record would have fields for the person's name, email address, and phone number.

For programming simplicity, LK has a syntax shortcut for accessing fields of a structure, which in actuality are the values of keyed entries in a table. The dot operator '.' transforms the text following the dot into a key index, as shown below.

```
person.name = "Jane Ruth";
person.email = "jane@lk.org";
person.phone = "0009997777";

outln(person.name); // prints "Jane Ruth"
outln(person); // prints { name=Jane Ruth email=jane@lk.org phone=0009997777 }

outln(person{"email"}); // prints "jane@lk.org"
outln(@person); // prints "name,email,phone"
```

You can build arrays and tables of structures. For example, a table of people indexed by their last names would make an efficient database for an address book.

```
db{"ruth"} = { "name"="Jane Ruth", "email"="jane@lk.org", "phone"="0009997777" };
db{"doe"} = { "name"="Joe Doe", "email"="joe@lk.org", "phone"="0008886666" };

// change joe's email address
db{"doe"}.email = "joe123@gmail.com";
outln( db{"ruth"}.phone );

// delete jane from the address book
db{"ruth"} = null;
```

6 Functions

It is usually good programming practice to split a larger program up into smaller sections, often called procedures, functions, or subroutines. A program may be easier to read and debug if it is not all thrown together, and you may have common blocks of functionality that are reused several times in the program.

A function is simply a named chunk of code that may be called from other parts of the script. It usually performs a well-defined operation on a set of variables, and it may return a computed value to the caller.

Functions can be written anywhere in a script, including inside other functions. If a function is never called by the program, it has no effect.

6.1 Definition

A function is defined by assigning a block of code to a variable. The variable is the name of the function, and the `define` keyword is used to create a new function.

Consider the very simple function listed below.

```
show_welcome = define()
{
   outln("Thank you for choosing LK.");
   outln("This text will be displayed at the start of the script.");
};
```

Notable features:

1. A function is created using the `define` keyword, and the result is assigned to a variable.
2. The variable references the function and can be used to call it later in the code.
3. The empty parentheses after the `define` keyword indicate that this function takes no parameters.
4. A code block enclosed by { } follows and contains the statements that execute when the function is called.
5. A semicolon finishes the assignment statement of the function to the variable `show_welcome`.

To call the function from elsewhere in the code, simply write the function variable's name, followed by the parentheses.

```
// show a message to the user
show_welcome();
```

6.2 Returning a Value

A function is generally more useful if it can return information back to the program that called it. In this example, the function will not return unless the user enters "yes" or "no" into the input dialog.

```
require_yes_or_no = define()
{
   while( true )
   {
      answer = in("Destroy everything? Enter yes or no:");
      if (answer == "yes")
          return true;
      if (answer == "no")
          return false;
      outln("That was not an acceptable response.");
   }
};

// call the input function
result = require_yes_or_no(); // returns true or false
if ( !result ) {
   outln("user said no, phew!");
   exit;
}  else  {
   outln("destroying everything...");
}
```

The important lesson here is that the main script does not worry about the details of how the user is questioned, and only knows that it will receive a `true` or `false` response. Also, the function can be reused in different parts of the program, and each time the user will be treated in a familiar way.

6.3 Parameters

In most cases, a function will accept parameters when it is called. That way, the function can change its behavior, or take different inputs in calculating a result. Analogous to mathematical functions, LK functions can take arguments to compute a result that can be returned. Arguments to a function are given names and are listed between the parentheses following the `define` keyword.

For example, consider a function to determine the minimum of two numbers:

```
minimum = define(a, b) {
   if (a < b) return a;
   else return b;
};
```

```
// call the function
count = 129;
outln("Minimum: " + minimum( count, 77) );
```

In LK, changing the value of a function's named arguments will modify the variable in the calling program. Instead of passing the actual value of a parameter `a`, a *reference* to the variable in the original program is used. The reference is hidden from the user, so the variable acts just like any other variable inside the function.

Because arguments are passed by reference (as in Fortran, for example), a function can "return" more than one value. For example:

```
sumdiffmult = define(s, d, a, b) {
   s = a+b;
   d = a-b;
   return a*b;
};

sum = -1;
diff = -1;
mult = sumdiffmult(sum, diff, 20, 7);

// will output 27, 13, and 140
outln("Sum: " + sum + " Diff: " + diff + " Mult: " + mult);
```

Functions can accept an unspecified number of arguments. Every named argument must be provided by the caller, but additional arguments can be sent to a function also. A special variable is created when a function runs called `__args` that is an array containing all of the provided arguments.

```
sum = define(init)
{
   for( i=1;i<#__args;i++ )
      init = init + __args[i];
   return init;
};

outln( sum(1,2,3,4,5) );  // prints 15
outln( sum("a","b","c","d") );  // prints abcd
```

6.4 Alternate Declaration Syntax

Functions can also be defined using the `function` keyword, as below. The syntax is simplified, and there is no semicolon required at the end because the definition does not comprise a statement in the usual sense. Functions declared in this way can be defined with parameters and can return values, as before.

```
function show_welcome(name)
{
```

```
   outln("Thank you " + name + " for choosing LK.");
   outln("This text will be displayed at the start of the script.");
}
```

In fact, this alternate syntax has an exact translation to the `define` syntax used elsewhere in this manual. For example:

```
function show_welcome( <args> )  {  <statements>  }
```

is exactly equivalent to

```
const show_welcome = define( <args> )  {  <statements>  };
```

The alternate syntax described here can help clarify programs and is generally easier for people familiar with other languages, at least initially. It automatically enforces *const-ness*, so that the function isn't inadvertently replaced later in the code by an assignment statement.

6.5 Environments and Scope

This sections reviews in technical detail the mechanisms in LK for storing information about data and function variables. The rules for variable access are simple but important to understand for writing script code that works as expected in all situations.

6.5.1 First-class Functions

Functions in LK are *first-class* values, meaning that they are referenced just as any other variable, and can be passed to other functions, replaced with new bodies, or defined implicitly. Functions can also be declared within other functions, stored in arrays, or referenced by keys in a table. The `define` keyword simply returns an internal LK representation of a function, and the assignment statement assigns that 'value' to the variable name. This ties the variable name to the function, implying that when the variable name is typed as a function call, LK knows to evaluate the function to which it refers.

```
triple              // variable name
   =                // assignment statement
     define( x )    // definition of a function with argument x
     {
       return 3*x;  // statements comprising function body
     }
     ;              // semicolon terminates the assignment statement

neg = define( y ) { return 0-y; }; // another function
```

In the example below, the `meta` function is unique that it returns another function. Which function it returns depends on the argument `mode`.

```
meta = define( mode ) {     // a function that returns another function
   if (mode == "double")
      return define(x) { return 2*x; };
```

```
   else
      return define(x) { return 3*x; };
};


outln( meta("triple")(12) );  // prints 36
outln( meta("double")(-23) ); // prints -46
```

It is important to note that the function returned by `meta` does not retain the context in which it was created. For example, if the body of the implicit function returned by `meta` referenced the `mode` argument in its calculations, running the returned function would result in an error because the `mode` argument would no longer be present in the current *environment*.

6.5.2 Environments

During the execution of a script, all variables (and thus functions) are stored in a special lookup table called the *environment*. The environment stores a reference from the variable name to the actual data or function represented by the variable, and allows new variables to be created and queried. When a script starts executing, the environment is empty, and as variables are assigned values, they are added to the environment. When an assignment statement changes the value of an existing variable, the old value is discarded and replaced with the new value.

6.5.3 Variable Lookup Mechanism

When a function is called, a new environment is created to store variables *local* to the function. The new environment retains a reference to its *parent environment*, which provides the mechanism by which variables and functions assigned outside a function can be accessed within it. LK follows these lookup rules:

1. The current environment is checked for the existence of the requested variable.
2. If it exists, the query is over and the variable has been found.
3. If the variable is not found, and the variable is referenced in a read-only or non-mutable context, LK checks all of the *parent environments* of the current environment to see if the variable has ever previously held a value visible in the environment hierarchy. If so, the variable has been found and the query is over. Otherwise, an error is thrown indicating an invalid access to an undefined variable.
4. If the variable is referenced in a mutable context (i.e. on the left hand side of an assignment statement), a new variable entry is created in the current (topmost or nearest) environment, even if a variable of the same name exists in a parent environment. This prevents variables outside of the function from being inadvertently updated. This behavior can be overridden using the `common` keyword, which explicitly allows a function to change a variable external to it.

In the script below, the variable `y` is not defined in any environment when it is first referenced in the body of the `triple` function. As a result, a new environment entry is created in the `triple` environment to hold its value, according to the lookup rules listed above.

```
triple = define (x) { y=3*x; return y; };
triple( 4 );
outln( y ); // this will fail because y is local to the triple function
```

In script below, the behavior is the same. The query for variable in `y` in the call to `triple( 4 )` will look in the environment created for the function call to `triple`, and, not finding `y`, will create a new entry named `y` because it is accessed in a mutable context. Since `y` is now local to the function call, the instance in the caller's context value will not be overwritten with the expression `3*x`, which in this context has value of 12.

```
y = 24;
// ... other code ...
triple = define (x) { y=3*x; return y; };
triple( 4 );
outln( y ); // this will print 24
```

6.5.4 The `common` Specifier

In the example above with the `triple` function, writing to variable `y` in the function call did not overwrite the external value previously defined in the caller's context. This default behavior helps avoid situations in which externally defined variables are inadvertently modified by a function call, which can become very difficult to debug.

However, in some programming situations, it may be desirable to use a function to change the variable defined in a parent environment. For this purpose, the `common` keyword is provided. It changes step 3 of the variable lookup rules to keep searching the hierarchy of environments when querying the existence of a variable, even when in a mutable or write context. The `common` specifier is typed before the name of a variable, and tells the LK engine that it is OK to modify the variable if it already exists outside the function. If it does not exist in any environment, the variable is created in the local context per the normal rules.

```
y = 24;
// ... other code ...
triple = define (x) { common y=3*x; return y; };
triple( 4 );
outln( y ); // this will print 12
```

More examples of the `common` keyword are listed below. Note that in most situations the `common` specifier is not needed and may be indicative of a sub-optimal program structure, and may introduce undesired behaviors that are difficult to trace and correct. In the example below, the counter variable `i` is declared `common` inside the `sum` function and so will have a different value after the `sum` function is called.

```
for (i=0;i<5;i++) outln("hello, " + i);
```

```
// ... other code ...

sum = define (x)
{
   sum=0;
   for (common i=0;i<#x;i++)
      sum = sum+x[i];
   return sum;
};

outln( sum( [ 1,2,3,4,5,6,7,8,9 ] )); // this will print 45
outln( i ); // prints 9
outln( sum );  // prints "<function>"
```

Arguments declared in the `define( )` operator are automatically created as local names in the function's environment.

6.6 Global Values

As we have seen, we can write useful functions using arguments and return values to pass data into and out of functions. However, sometimes there are some many inputs to a function that it becomes very cumbersome to list them all as arguments. Alternatively, you might have some variables that are used throughout your program, or are considered reference values or constants. Because of the hierarchical environment query rules, variables declared in the *main* body of the script can be treated as global values.

```
m_pi = 3.1415926;  // treat this variable as a global

circumference = define( r ) { return 2*m_pi*r; };
deg2rad = define( x ) { return m_pi/180*x; };

outln( "PI: " + m_pi );
outln( "CIRC: " + circumference( 3 ) );
outln( "D2R: " + deg2rad( 90 ) );
```

Common programming advice is to minimize the number of global variables used in a program. Sometimes the are certainly necessary, but too many can lead to mistakes that are harder to debug and correct, and can reduce the readability and maintainability of your script.

6.7 Objects and Methods

Using functions and tables, it is possible to create objects in LK that encapsulate functionality in a reusable unit. LK does not explicitly support classes and inheritance like true object-oriented languages like C++ and Java, but clever use of the facilities described thus far in this manual can go a long way in providing the user with powerful capabilities.

An object can be defined in LK by writing a function that acts like a *constructor* to create a table with various fields. Some of the fields may be data, but others may actually be functions that operate on the data fields of the table. This is possible because functions are *first-class* values.

Note the use of a special variable called `this` in functions defined as fields in the table (`area`, `perim`, and `scale_side`). `this` is created and assigned automatically if the function is invoked using the `->` operator, discussed below.

```
make_square = define(side)
{
   local obj.label = "Square";
   obj.side = side;

   obj.area = define() { return this.side*this.side; };
   obj.perim = define() { return 4*this.side; };
   obj.scale_side = define(factor)
   {
      this.side = this.side * factor;
      return this.side;
   };

   return obj;
};


sq[0] = make_square(2);
sq[1] = make_square(4);
sq[2] = make_square(5);

outln( sq[0].label ); // prints "Square"
outln( sq[1].perim ); // prints <function>
outln( sq[1].side );  // prints 4
for (i=0;i<#sq;i++)
  outln("sq " + i + " area: " + sq[i]->area() + " perim: " + sq[i]->perim());


outln("new side: " + sq[0]->scale_side(3));
outln("area: " + sq[0]->area());
outln("new side: " + sq[0]->scale_side(1/3));
outln("area: " + sq[0]->area());
```

In this example, the `make_square` function works like a *constructor* to create a new object of type square, implemented internally as a table. The `area` and `perim` functions defined within the constructor both take a parameter called `this`, which is a reference to an object created with `make_square`. Once all of the fields and methods of the square object are defined, the object is returned. When we say object, in fact we are simply referring internally to a table, whose fields are the memb

We then create two instances of a square as two elements in the array *sq*. In the `for` loop that prints information about each square in the array, the member functions are called using the `->` operator. The `->` is the *thiscall* operator. It implicitly passes the left hand side of the operator to the function, in a special local variable called `this`. For example, in the `scale_side` method, the variable `this` variable is used to access the data members of the object.

6.8 Built-in Functions

Throughout this guide, we have made use of built-in functions like `in`, `outln`, and others. These functions are included from the LK standard library automatically, and called in exactly the same way as user functions. Like user functions, they can return values, and sometimes they modify the arguments sent to them. Refer to the "Standard Library Reference" at the end of this guide for documentation on each function's capabilities, parameters, and return values.

7 Input, Output, and System Access

LK provides a variety of standard library functions to work with files, directories, and interact with other programs. So far, we have used the `in` and `outln` functions to accept user input and display program output in the runtime console window. Now we will learn about accessing files and other programs.

7.1 Working with Text Files

To write data to a text file, use the `write_text_file` function. `write_text_file` accepts any type of variable, but most frequently you will write text stored in a string variable. For example:

```
data = "";
for (i=0;i<10;i=i+1) data = data + "Text Data Line " + to_string(i) + "\n";
ok = write_text_file( "C:/test.txt", data );
if (!ok) outln("Error writing text file.");
```

Reading a text file is just as simple with the `readtextfile` function. If the file is empty or cannot be read, an empty text string is returned.

```
mytext = read_text_file("C:/test.txt");
out(mytext);
```

While these functions offer an easy way to read an entire text file, often it is useful to be able to access it line by line. The `open`, `close`, and `readln` functions are for this purpose.

```
file = open("c:/test.txt", "r");
if (!file) {
   outln("could not open file");
   exit;
}
```

```
line="";
while ( read_line( file, line ) ) {
   outln( "My Text Line='" + line + "'" );
}

close(file)
```

In the example above, `file` is a number that represents the file on the disk. The `open` function opens the specified file for reading when the `"r"` parameter is given. The `read_line` function will return `true` as long as there are more lines to be read from the file, and the text of each line is placed in the `line` variable.

Another way to access individual lines of a text file uses the `split` function to return an array of text lines. For example:

```
lines = split( read_text_file( "C:/test.txt" ), "\n" );
outln("There are " + #lines + " lines of text in the file.");
if (#lines > 5) outln("Line 5: '", lines[5], "'");
```

7.2 File System Functions

Suppose you need to process many different files, and consequently need a list of all the files in a folder that have a specific extension. LK provides the `dir_list` function to help out in this situation. If you want to filter for multiple file extensions, separate them with commas.

```
list = dir_list( "C:/Windows", "dll" ); // could also use "txt,dll"
outln("Found " + #list + " files that match.");
outln(join(list, "\n"));
```

To list all the files in the given folder, leave the extension string empty or pass `"*"`.

Sometimes you need to be able to quickly extract the file name from the full path, or vice versa. The functions `path_only`, `file_only`, `ext_only` extract the respective sections of a file name, returning the result.

To test whether a file or directory exist, use the `dir_exists` or `file_exists` functions. Examples:

```
path = "C:/SAM/2010.11.9/samsim.dll";
dir = path_only( path );
name = file_only( path );
outln( "Path: " + path );
outln( "Extension: " + ext_only(path));
outln( "Name: " + name + " Exists? " + file_exists(path) );
outln( "Dir: " + dir + " Exists? " + dir_exists(dir));
```

7.3 Calling Other Programs

Suppose you have a program on your computer that reads an input file, makes some complicated calculations, and writes an output file. The `system` function executes an external program, and returns the integer result code, after waiting for the called program to finish. Examples:

```
system("notepad.exe"); // run notepad and wait
```

Each program runs in a folder that the program refers to as the *working directory*. Sometimes you may need to switch the working directory to conveniently access other files, or to allow an external program to run correctly. The `cwd` function either gets or sets the current working directory.

```
working_dir = cwd();    // get the current working directory
cwd( "/usr/local/bin" );    // change the working directory
outln("cwd=" + cwd() );
cwd(working_dir);       // change it back to the original one
outln("cwd=" + cwd() );
```

8 Writing Extensions

LK provides an application programming interface (API) for writing libraries of functions that can be loaded at runtime into the scripting engine. Dynamically loaded libraries can be used to provide very high performance computations, ability to create linkages to other software packages, and to integrate existing codes written in other languages. The C language extension API is fully defined in the header file `lk_invoke.h`, and does require the extension to be linked to any other LK library or otherwise. As a result, programming, compiling, and linking a standalone LK extension is very straightforward.

Note: Currently, the LK extension API does not support passing Unicode text back and forth between an extension and the core LK engine, even if the core engine is compiled utilizing a Unicode string class.

8.1 Basic Organization

The examples in this manual show how to construct an LK extension in C or C++. First, the `lk_invoke.h` header file must be included in the C source file. This defines several functions and macros to assist writing functions that can be dynamically loaded by LK. In the very simple example extension below, a single function is defined that returns a constant value, the speed of light.

```
#include <lk_invoke.h>

LK_FUNCTION( speed_of_light )
{
   LK_DOCUMENT( "speed_of_light", "Returns the speed of light in (m/s).", "(none):number" );

   lk_return_number( 299792458 );
```

```
}

LK_BEGIN_EXTENSION()
  speed_of_light
LK_END_EXTENSION()
```

To define a new function, use the `LK_FUNCTION( name )` macro, called with the name of the function. This macro expands to the C code `void name( struct __lk_invoke_t *lk )`.

The first line of any extension function must be the documentation specifier, defined by the macro `LK_DOCUMENT( name, description, signature )`. This macro takes three arguments: the name of the function by which LK will refer to it, a text description of what it does, and a *signature* that shows the proper parameters and return value. For functions that may take multiple forms (i.e. different behavior and signatures), the `LK_DOCUMENT2(...)` and `LK_DOCUMENT3(...)` macros exist. These macros take multiple descriptions and signatures, and are fully explained in the `lk_invoke.h` header file.

To make the functions visible to the LK engine, they must be listed between the `LK_BEGIN_EXTENSION()` and `LK_END_EXTENSION()` macros. Otherwise, the functions will not be properly exported from the dynamic library. These macros may only appear once within the source code of an extension library.

8.2 Working with Variables

The extension API defines several macros for easily working with function arguments and return values. A variable is referenced by the opaque pointer type `lk_var_t`, and is passed as an argument to the various functions to retrieve and set values. The `sum` example below shows how to work with function arguments and return values explicitly, as well as how to signal an error if necessary.

```
LK_FUNCTION( sum )
{
   LK_DOCUMENT( "sum", "Sums up all the parameters passed.", "(...):number" );

   double val = 0;
   int i;
   int count = lk_arg_count();
   if (count < 1)
   {
      lk_error("sum() must be provided more than zero arguments");
      return;
   }

   for (i=0;i<count;i++)
   {
      lk_var_t x = lk_arg( i );
      val = val + lk_as_number( x );
   }
```

```
   lk_var_t ret = lk_result();
   lk_set_number( ret, val );
}
```

The extension API also provides facilities for working with arrays and tables. In the example below, the array passed in is converted to a table with named indices.

```
LK_FUNCTION( tabulate )
{
   LK_DOCUMENT( "tabulate", "Converts an array to a table.", "(array):table");

   lk_var_t arr, ret;
   int len, i;
   char key[64];

   if (lk_arg_count() != 1
      || lk_type( lk_arg(0) ) != LK_ARRAY)
   {
      lk_error("tabulate() requires one array parameter");
      return;
   }

   ret = lk_result();
   lk_make_table(ret);

   arr = lk_arg(0);
   len = lk_length( arr );

   for (i=0;i<len;i++)
   {
      sprintf(key, "item%d", i);

      lk_var_t item = lk_index(arr, i);
      int ty = lk_type(item);
      switch(ty)
      {
      case LK_NUMBER:
         lk_table_set_number( ret, key, lk_as_number( item ) );
         break;
      case LK_STRING:
         lk_table_set_string( ret, key, lk_as_string( item ) );
         break;

      case LK_ARRAY:
      case LK_TABLE:
         lk_error("arrays and tables not currently copied. exercise for the user!");
```

```
         return;
      }
   }
}
```

Running the following LK script code after loading a compiled extension with the `tabulate()` function above shows that the array has been converted to a table with items `item0`, `item1`, and so on.

```
arr = [ 1, 5, 'h3llo', '213', 451.4, -1 ];
x = tabulate(arr);
outln(x);
```

Output:

```
{ item2=h3llo item3=213 item0=1 item1=5 item4=451.4 item5=-1 }
```

For additional variable manipulation macros, refer to the `lk_invoke.h` header file. Additional advanced facilities not documented in this manual are also available for implementing callback-type functions and object referencing.

8.3 Compiling and Loading

Using the MinGW compiler toolchain on Windows, it is very straightforward to create an LK extension library. Simply issue the command below at the Windows command prompt (`cmd.exe` from Start/Run)

```
c:\> gcc -shared -o testextension.dll testextension.c
```

To load this extension at runtime into a LK script, use the standard library function `load_extension()`, per below. If the extension loads successfully, any of the functions defined and exported in the extension can be invoked just like any other built-in or user-defined LK function.

```
ok = load_extension( "c:/testextension.dll" );
if (!ok) {
   // failed to load extension
   exit;
}
c = speed_of_light();
```

The dynamic extension library will be unloaded when the script finishes executing.

9 Standard Library Reference

The standard library documented here outlines built-in functions that are likely to be available in any implementation of the LK scripting language. The input/output functions referenced in this manual (`in(), out(), outln()`) are not part of the standard library because the LK host

environment defines how these functions are manifested. It is assumed, however, that most host environments will include these "standard" I/O functions as well.

9.1 Standard System Functions

`to_int`(any):integer
Converts the argument to an integer value.

`to_real`(any):real
Converts the argument to a real number (double precision).

`to_bool`(any):boolean
Converts the argument to a boolean value (1=true, 0=false).

`to_string`([any]):string
Converts the argument[s] to a text string.

`alloc`(integer, {integer}):array
Allocates an array of one or two dimensions.

`dir_list`(string:path, string:extensions, [boolean:list dirs also]):array
List all the files in a directory. The extensions parameter is a comma separated list of extensions, or * to retrieve every item.

`file_exists`(string):boolean
Determines whether a file exists or not.

`dir_exists`(string):boolean
Determines whether the specified directory exists.

`rename_file`(string:old name, string:new name):boolean
Renames an existing file.

`remove_file`(string):boolean
Deletes the specified file from the filesystem.

`mkdir`(string:path, {boolean:make_full=true}):boolean
Creates the specified directory, optionally creating directories as need for the full path.

`path_only`(string):string
Returns the path portion of a complete file path.

`file_only`(string):string
Returns the file name portion of a complete file path, including extension.

`ext_only`(string):string
Returns the extension of a file name.

`cwd`(void):string
Returns the current working directory.

`cwd`(string):boolean
Sets the current working directory.

Notes: Two modes of operation: set or retrieve the current working directory.

`system`(string):integer
Executes the system command specified, and returns the exit code.

`write_text_file`(string:file, string:data):boolean
Writes data to a text file.

`read_text_file`(string:file):string
Reads the entire contents of the specified text file and returns the result.

`open`(string:file, string:rwa):integer
Opens a file for 'r'eading, 'w'riting, or 'a'ppending.

`close`(integer:filenum):void
Closes an open file.

`seek`(integer:filenum, integer:offset, integer:origin):integer
Seeks to a position in an open file.

`tell`(integer:filenum):integer
Returns the current value of the file position indicator.

`eof`(integer:filenum):boolean
Determines whether the end of a file has been reached.

`flush`(integer:filenum):void
Flushes the file output stream to disk.

`read_line`(integer:filenum, @string:buffer):boolean
Reads one line of text from a file, returning whether there are more lines to read.

`write_line`(integer:filenum, string:dataline):boolean
Writes one line of text to a file.

`read`(integer:filenum, @string:buffer, integer:bytes):boolean
Reads data from a file, returning false if there is no more data to read.

`write`(integer:filenum, string:buffer, integer:bytes):boolean
Writes data to a file.

`load_extension`(string:path):boolean
Loads an LK extension library.

`extensions`(none):table
Returns information about currently loaded extensions

`ostype`(none):string
Returns identifying information about the operating system type. ('osx', 'win32', 'linux', etc).

9.2 String Functions

`sprintf`(string:format, ...):string
Returns a formatted string using standard C printf conventions, but adding the %m and %, specifiers

for monetary and comma separated real numbers.

`strlen`(string):integer
Returns the length of a string.

`strpos`(string, string):integer
Locates the first instance of a character or substring in a string.

`first_of`(string:s1, string:s2):integer
Searches the string s1 for any of the characters in s2, and returns the position of the first occurence

---

`last_of`(string:s1, string:s2):integer
Searches the string s1 for any of the characters in s2, and returns the position of the last occurence

---

`left`(string, integer:n):string
Returns the leftmost n characters of a string.

`right`(string, integer:n):string
Returns the rightmost n characters of a string.

`mid`(string, integer:start, {integer:length}):string
Returns a subsection of a string.

`ascii`(character):integer
Returns the ascii code of a character.

`char`(integer):string
Returns a string (one character long) from an ascii code.

`ch`(string, integer, character):void
Sets the character at the specified index in a string.

`ch`(string, integer):character
Gets the character at the specified index in a string.

Notes: Two modes of operation: for retrieving and setting individual characters in a string.

`isdigit`(character):boolean
Returns true if the argument is a numeric digit 0-9.

`isalpha`(character):boolean
Returns true if the argument is an alphabetic character A-Z,a-z.

`isalnum`(character):boolean
Returns true if the argument is an alphanumeric A-Z,a-z,0-9.

`upper`(string):string
Returns an upper case version of the supplied string.

`lower`(string):string
Returns a lower case version of the supplied string.

`replace`(string, string:s1, string:s2):string
Replaces all instances of s1 with s2 in the supplied string.

`split`(string:s, string:delims, {boolean:ret_empty=false}, {boolean:ret_delim=false}):array
Splits a string into parts at the specified delimiter characters.

`join`(array, string:delim):string
Joins an array of strings into a single one using the given delimiter.

`real_array`(string):array
Splits a whitespace delimited string into an array of real numbers.

9.3 Math Functions

`ceil`(real:x):real
Round to the smallest integral value not less than x.

`floor`(real:x):real
Round to the largest integral value not greater than x.

`sqrt`(real:x):real
Returns the square root of a number.

`pow`(real:x, real:y):real
Returns a number x raised to the power y.

`exp`(real:x):real
Returns the base-e exponential of x.

`log`(real:x):real
Returns the base-e logarithm of x.

`log10`(real:x):real
Returns the base-10 logarithm of x.

`pi`(void):real
Returns the value of PI.

`sgn`(real:x):real
Returns 1 if the argument is greater than zero, 0 if argument is 0, otherwise -1.

`abs`(real:x):real
Returns the absolute value of a number.

`sin`(real:x):real
Computes the sine of x (radians)

`cos`(real:x):real
Computes the cosine of x (radians)

`tan`(real:x):real
Computes the tangent of x (radians)

---

asin(real:x):real
Computes the arc sine of x, result is in radians, -pi/2 to pi/2.

---

acos(real:x):real
Computes the arc cosine of x, result is in radians, 0 to pi.

---

atan(real:x):real
Computes the arc tangent of x, result is in radians, -pi/2 to pi/2.

---

atan2(real:x, real:y):real
Computes the arc tangent using both x and y to determine the quadrant of the result, result is in radians.

---

sind(real:x):real
Computes the sine of x (degrees)

---

cosd(real:x):real
Computes the cosine of x (degrees)

---

tand(real:x):real
Computes the tangent of x (degrees)

---

asind(real:x):real
Computes the arc sine of x, result is in degrees, -90 to 90.

---

acosd(real:x):real
Computes the arc cosine of x, result is in degrees, 0 to 180.

---

atand(real:x):real
Computes the arc tangent of x, result is in degrees, -90 to 90.

---

atan2d(real:x, real:y):real
Computes the arc tangent using both x and y to determine the quadrant of the result, result is in degrees.

`nan`(void):real
Returns the non-a-number (NAN) value.

`isnan`(number):boolean
Returns true if the argument is NaN.

`mod`(integer:x, integer:y):integer
Returns the remainder after integer division of x by y.

`sum`(...):real
Returns the numeric sum of all values passed to the function. Arguments can be arrays or numbers.

`min`(...):real
Returns the minimum of the numeric arguments.

`min`(array):real
Returns the minimum value in an array of numbers

Notes: Returns the minimum numeric value.

`max`(...):real
Returns the maximum of the passed numeric arguments.

`max`(array):real
Returns the maximum value in an array of numbers

Notes: Returns the maximum numeric value.

`mean`(...):real
Returns the mean (average) value all values passed to the function. Arguments can be arrays or numbers.

`stddev`(...):real
Returns the sample standard deviation of all values passed to the function. Uses Bessel's correction (N-1). Arguments can be arrays or numbers.

`gammaln`(real):real
Computes the logarithm of the Gamma function.

---

pearson(array:x, array:y):real
Calculate the Pearson linear rank correlation coefficient of two arrays.

---

besj0(real:x):real
Computes the value of the Bessel function of the first kind, order 0, J0(x)

---

besj1(real:x):real
Computes the value of the Bessel function of the first kind, order 1, J1(x)

---

besy0(real:x):real
Computes the value of the Bessel function of the second kind, order 0, Y0(x)

---

besy1(real:x):real
Computes the value of the Bessel function of the second kind, order 1, Y1(x)

---

besi0(real:x):real
Computes the value of the modified Bessel function of the first kind, order 0, I0(x)

---

besi1(real:x):real
Computes the value of the modified Bessel function of the first kind, order 1, I1(x)

---

besk0(real:x):real
Computes the value of the modified Bessel function of the second kind, order 0, K0(x)

---

besk1(real:x):real
Computes the value of the modified Bessel function of the second kind, order 1, K1(x)

---

erf(real):real
Calculates the value of the error function

---

erfc(real):real
Calculates the value of the complementary error function