Paper SAS1561-2015

Driving SAS® with Lua

Paul Tomas, SAS Institute Inc.

ABSTRACT

Programming SAS® has just been made easier, now that SAS 9.4 has incorporated the Lua programming language into the heart of the SAS System. With its elegant syntax, modern design, and support for data structures, Lua offers you a fresh way to write SAS programs, getting you past many of the limitations of the SAS macro language. This paper shows you how you can get started using Lua to drive SAS, via a quick introduction to Lua and a tour through some of the features of the Lua and SAS combination that make SAS programming easier. SAS macro programming is also compared with Lua, so that you can decide where you might benefit most from using each language.

INTRODUCTION

Base SAS® 9.4 introduces the LUA procedure which embeds the Lua 5.2 run-time environment into the SAS System and makes SAS functionality available through Lua. Lua offers a simple syntax and helpful error messages, and it supports data structures and modules. All these Lua features contribute to making your SAS programming easier, especially for larger and more complex projects. Lua is a programming language best known for its excellent performance both in terms of speed and memory footprint. Lua was created by Roberto Ierusalimschy at the Pontifical Catholic University in Brazil and is maintained by a small group of developers there. Lua's purpose is to script C-based software. Because SAS is written in C, Lua is a great fit for SAS. Lua is distributed as open-source software and is documented both online and in a number of books. Lua has been ported to a number of platforms, including iOS and Android. Some popular games, such as *Angry Birds*, are written in Lua.

Lua is not a replacement for the SAS DATA step or any of the SAS procedures, but rather a tool that will enhance your ability to drive SAS by using some of the powerful capabilities of a modern programming language.

Lua's simple syntax makes it easy to learn by example. But beyond the basics, Lua provides some additional constructs that you can use to capture some of the more advanced concepts behind the language. You are encouraged to consult the resources mentioned at the end of this paper, in particular Ierusalimschy (2013).

In this paper, inline Lua keywords and code phrases use bold Courier font (for example, "the **`print`** function"), whereas inline SAS keywords use plain font in all capital letters (for example, "the TRIM function").

GETTING STARTED

Many introductory texts to a programming language start off with a "Hello, world" example. This classic greeting program can be written in Lua in just one line:

```
print("Hello, world")
```

Line comments in Lua begin with **`--`**, and block comments begin with **`--[[`** and end with **`--]]`**. So the program could be embellished with documentation as follows:

```
--[[
   This program prints 'Hello, world' to the console.
--]]
print("Hello, world")  -- print 'Hello, world'
```

Notice that statements do not end with semicolons (although you can add semicolons without any ill effects). However, you do need semicolons if you write multiple statements on one line:

```
print("Press the button."); print("That was easy!")
```

RUNNING Lua IN SAS

The LUA procedure enables you to run Lua code from SAS. PROC LUA runs the Lua virtual machine inside the SAS process to offer seamless integration of Lua with SAS. You can execute Lua code by embedding it within a SUBMIT/ENDSUBMIT block in the PROC LUA invocation, as shown in the following statements:

```
/* Embedded Lua statements */
proc lua ;
 submit;
   print('Hello from Lua!')
 endsubmit;
run;
```

The preceding program, prints the **'Hello from Lua!'** message to the SAS log. PROC LUA routes output from Lua's print function to the SAS log.

Lua BASICS

Lua VARIABLES

In Lua, you usually declare a variable by using a **local** statement. For example:

```
local pi = 3.14          -- Declare a numeric variable
local guest = "Veronica" -- Declare a string variable
```

Using the **local** keyword is recommended because omitting it results in the variable being declared to be global in scope.

Lua infers the type of the variable from the value that is provided to it; hence there is no type qualifier in variable declarations. The basic types are *number*, *string*, and *boolean*. The Boolean values are **true** and **false**. There is also a special type called *nil*, which is used to represent the absence of a value and is treated as **false** when evaluated in a Boolean context. In Lua, any value that is not **nil** or **false** is evaluated as **true** in a Boolean context. This includes the numeric value zero which, although interpreted as **false** in many other languages, is treated as Boolean **true** in Lua.

SAS missing values need special treatment because they are not like ordinary numbers. Currently only the standard SAS missing value (**.**) is supported in Lua. Missing values are discussed in the section "MISSING VALUES" on page 6.

A number of string manipulation functions are available to extract substrings, locate character sequences, replace fragments, and so on. One common string operator is **'..'**, which is used to concatenate strings together, as shown in the following statements:

```
local str1 = "Hello"
local str2 = "world"
print(str1 .. " " .. str2) -- prints "Hello world"
```

Other variable types exist, such as **table** (see the next section) and **function**. For more information about these and other types, see Ierusalimschy (2013) and the Lua documentation at http://www.lua.org.

TABLES

Lua also supports flexible in-memory data structures called *tables*, which can be extremely useful and can drastically simplify your programming.

Tables as Arrays

Tables can be set up to act like arrays (lists), as in the following statement:

```
local shoppingList = {'milk', 'flour', 'eggs', 'sugar'}
```

When tables are declared in this fashion, items are indexed by a numeric index:

```
local drink = shoppingList[1]
```

Much like SAS DATA step arrays, arrays in Lua start at index 1 by default.

You can iterate over an array by using the **ipairs** function, as in the following code:

```
for i, item in ipairs(shoppingList) do
   print(i, item)
end
```

This code prints the following to the console:

```
1       milk
2       flour
3       eggs
4       sugar
```

The **ipairs** function returns two values with each iteration: the current index into the array (assigned to variable **i** in the preceding example) and the value of the item at that index in the array (assigned to variable **item**).

Tables as Dictionaries

Lua tables can also be set up to act like dictionaries (hash tables). For example:

```
local band = {
             vocals='Robert Plant',
             guitar='Jimmy Page',
             bass='John Paul Jones',
             drummer='John Bonham'
           }
```

Using a table in this manner enables you to index entries by key, as in the following statement:

```
local singer = band.vocals  -- you can also use band['vocals']
```

Lua provides a function called **pairs** (similar to **ipairs**) that iterates through all the key/value combinations in the dictionary. For example, the following statements print each key and value that are found in the previously declared Lua table:

```
for key, value in pairs(band) do
 print(key, value)  -- print each key and value found in the 'band' table
end
```

The preceding code produces the following output:

```
bass    John Paul Jones
drummer John Bonham
vocals  Robert Plant
guitar  Jimmy Page
```

Tables can contain other tables, and tables can also contain functions (as described in the next section). The combination of array-like behavior with hash table semantics enables you to represent pretty much any type of data structure in memory. Furthermore, Lua provides the concept of *metatables*, which enable you to control how tables respond to the insertion and retrieval of values. For more information about Lua tables, see Ierusalimschy (2013).

FUNCTIONS

Functions can be defined in Lua by using the **`function`** keyword, as follows:

```
function sayHello(name)
   print('Hello ' .. name)
end
```

Function arguments are declared within the set of parentheses that precedes the function body, and they are positional only. In the preceding example, **`name`** is the one and only argument to **`sayHello`**. So the function could be invoked via the following statement:

```
sayHello('Veronica')
```

You can achieve the effect of named arguments by using a Lua table. For example, the following statements declare a function that expects a single table parameter (**`args`**). The function accesses the elements of that table by key name (**`firstName`**, **`lastName`**). The function is then invoked by passing in a Lua table that has values for those keys.

```
function sayHello(args)
 print('Hello ' .. args.firstName .. ' ' .. args.lastName)
end
sayHello({ firstName='Andrew', lastName='Carnegie'})
```

The preceding code prints the following output:

```
Hello Andrew Carnegie
```

In Lua, you can assign functions to variables, just as you would assign numbers, strings, and tables. This is especially useful in creating tables of functions. Lua *modules* are implemented this way, with functions assigned to tables, using keys for the function names. For example, the following statements create a table called **`my_module`** that contains two functions, **`sayHello`** and **`sayGoodBye`**:

```
local my_module = {}
my_module.sayHello = function(name)
                      print("Hello " .. name)
                     end
my_module.sayGoodBye = function(name)
                        print("Goodbye " .. name)
                       end
```

Now that this table is defined, you can access the functions just as you would access any other type of table entry. For example:

```
my_module.sayHello("Dave")
my_module.sayGoodBye("Helen")
```

You return values from functions by using a **return** statement. For example, the following function definition returns the result of calculating the area of a circle, given the radius as the function argument:

```
function area_of_circle(radius)
    return 3.14*radius^2
end
```

Lua enables you to return multiple values from a function—the **ipairs** function is an example of this. To write a function that returns multiple values, you simply provide those values, separated by commas, to the **return** statement. The following example makes use of some functions that are provided via the **sas** table (discussed in the next section) to retrieve the components of the current date:

```
function split_date()
   -- get current date
   local date = sas.date()
   -- return day, month, year
   return sas.day(date), sas.month(date), sas.year(date)
end
```

You could then call this function as follows:

```
local day, month, year  = split_date()
print("d=", d, "m=", m, "y=",y)
```

Now that the basics of Lua have been covered, the next step is to see how you can use Lua in your SAS programs.

CALLING SAS FUNCTIONS FROM Lua

THE sas TABLE

When PROC LUA initializes the Lua state, it creates a special global Lua table called **sas**. This table is always available to your Lua programs. All the SAS functions you are accustomed to using in DATA step code (such as MIN, MAX, TRIM, and so on) are available in Lua by adding the prefix **sas.** to their names. So the MIN function in SAS becomes **sas.min** in Lua, the TRIM function in SAS becomes **sas.trim** in Lua, and so on. The only exceptions to this rule pertain to functions that make sense to operate only in a SAS DATA step. One example is the LAG function, which has no equivalent outside a DATA step and therefore cannot be called from Lua.

The following code uses the **sas.symget** function to read a SAS macro variable and the **sas.symput** function to update that same macro variable:

```
%let foo=bar;
proc lua;
submit;
  local foo = sas.symget("foo")
  print("foo is ", foo)  -- prints 'bar'
  sas.symput('foo','baz')
endsubmit;
run;
%put &foo; /* prints 'baz' */
```

The call to **sas.symput** updates the macro variable **foo** with the new value **baz**. The %PUT statement that follows the PROC LUA invocation picks up the new value for **foo**.

The various math functions provided by Lua (such as **pow**, **sin**, **cos**, and so on) are mapped to their SAS equivalents by PROC LUA so that your results stay consistent.

CALLING FCMP FUNCTIONS

You can also call functions that are created by the FCMP procedure. Only OUTARGS arguments that are arrays can be modified by the called function. To use FCMP functions from Lua, simply define them as you normally would by using PROC FCMP, and then set the CMPLIB= system option. You can then call your new functions from Lua as you would for any other SAS function, by using the same **sas.** prefix. The following statements use the FCMP procedure to define a function called SUMX, which is subsequently accessed from Lua by referring to it using the name **sas.sumx**:

```
proc fcmp outlib=work.foo.foo;
/* define new function called 'sumx' */
function sumx(x[*]);
  sum = 0;
  do i = 1 to dim(x);
     sum = sum + x[i];
  end;
  return(sum );
endsub;
run;
options cmplib=work.foo;
proc lua ;
 submit;
  array = { 1, 2, 3, 4, 5 }
  sum = sas.sumx(array)  -- call 'sumx' from Lua
  print(sum)
 endsubmit;
run;
```

MISSING VALUES

As mentioned earlier, SAS missing values need special treatment. You can check whether a value is missing by using the **sas.is_missing** function, or just by using Lua's equality operator **==** against the **sas.MISSING** value as follows:

```
local val = sas.inputn(".", "2.")
print(val==sas.MISSING)  -- or you can use sas.is_missing(val)
```

In Lua, comparisons with other numbers work as expected, with **sas.MISSING** interpreted as negative infinity.

SUBMITTING SAS CODE FROM Lua

In addition to providing a gateway to SAS functions, the **sas** table contains some additional functions to make integration with SAS easier. One such function is the **sas.submit** function, which enables you to submit SAS code from Lua. The following example calls **sas.submit** to sort a data set:

```
sas.submit("proc sort data=work.mydata; run;")
```

Using Lua's support for **[[** and **]]** as text delimiters, you could also write the preceding code as follows:

```
sas.submit[[
  proc sort data=work.mydata;
  run;
]]
```

Using **[[** and **]]** delimiters enables you to use quotes and double quotes within the block of code without needing to escape those characters.

Local variables that precede the call to **sas.submit** are automatically resolved in the block of code if they are referenced through the use of **@** delimiters, as in the following:

```
local dataset = "sashelp.class"
sas.submit[[
     proc print data=@dataset@;
     run;
]]
```

The **sas.submit** function also supports an optional table parameter so that you can explicitly control the key-value pairs that are made available for resolution in the block of SAS code. The following statements invoke **sas.submit** by passing in two arguments: the block of SAS code (as a Lua string), and a Lua table containing a key-value pair that is used to resolve the **@dataset@** token in the SAS code:

```
sas.submit([[
       proc print data=@dataset@;
       run;
]], { dataset="sashelp.class"})
```

The preceding invocation to **sas.submit** causes the following SAS code to be submitted:

```
proc print data=sasehelp.class;
run;
```

sas.submit_ FUNCTION

Sometimes you might want to submit some SAS code but not execute it immediately, such as when you are generating SAS code in a loop and you need to finish up with a submitted RUN statement. In this case, you use the **sas.submit_** function (that is **sas.submit** followed by an underscore). For example:

```
sas.submit_([[ data work.class; ]]
:
-- more submit_ calls
:
sas.submit("run;")
```

The **sas.submit_** function queues the SAS code for execution, but that code does not execute until you follow up with a call to **sas.submit**.

READING AND WRITING SAS DATA SETS

Submitting SAS code is useful, but often the control logic that determines what SAS code needs to be submitted next depends on the values that are contained in a SAS data set. To read values from a SAS data set, you use the **sas.open** function, which returns a handle that provides functions to read observations, retrieve values of data set variables, and release the handle.

The following simple example opens a data set and prints some values from each observation:

```
local dsid = sas.open("sashelp.class")
for row in sas.rows(dsid) do
  print(row.name, row.age)
end
sas.close(dsid)
```

Each row that is returned by the **sas.rows()** function in the **for** loop is returned as a Lua table, using the data set variable name in lower case as the key that corresponds to each value in that row. After reading each row, the **sas.close** function is called to release the resources that are associated with the data set handle. If you forget to call **sas.close()** after using the data set, you will see a warning like the following printed to SAS log when the data set handle goes out of scope:

```
WARNING: Closing SASHELP.CLASS - handle has gone out of scope.
```

This is Lua's garbage collection mechanism at work, making sure that unused variable references are cleaned up.

Although the preceding example works well for narrow tables that contain relatively few columns, it can be expensive for wider tables, because each variable is preloaded into the Lua table for each row. If you don't need to access all variables at one time for each observation, you can instead select the variables you want by using the **sas.get** function, as follows:

```
local dsid = sas.open("sashelp.class")
while sas.next(dsid) do
  print(sas.get(dsid,"name"), sas.get(dsid,"age"))
end
sas.close(dsid)
```

The **sas.next()** function moves forward through a SAS data set until it reaches the last observation. After it reaches the last observation, it returns **nil**, which evaluates to **false**, thus ending the **while** loop. The **sas.get** function is used to read values from the current observation; it accepts either a variable name as a string value or the variable index (position) as a number. For example, an alternate method to read the values of every data set variable could be implemented as follows:

```
local dsid = sas.open("sashelp.class")
local nvars = sas.nvars(dsid)
while sas.next(dsid) do
   for i=1,nvars do
      print(sas.get(dsid,i))
   end
end
sas.close(dsid)
```

Here the **sas.nvars** function is used to determine the number of variables in the data set.

The **sas.vars** function, similar to the **sas.rows** function, enables you to iterate across metadata for all the variables in a SAS data set, as in the following statements:

```
local dsid = sas.open("sashelp.class")
for var in sas.vars(dsid) do
   print("var=", table.tostring(var))
end
sas.close(dsid)
```

The value returned by each call to **sas.vars** is a Lua table that contains the attributes of the current variable. If you were to run that example, you would see entries like the following printed to the SAS log:

```
var=    table: 0AF61828=
{
  ["fmt_dec"]=0
  ["type"]="C"
  ["label"]=""
  ["fmt_width"]=0
  ["informat"]=""
  ["name"]="Name"
  ["infmt_dec"]=0
  ["format"]=""
  ["infmt_width"]=0
  ["length"]=8
}
 :
```

If, instead of iterating across all variables, you only want to access information about a single variable, you can do so by using the **sas.varinfo** function as follows:

```
local dsid = sas.open("sashelp.class")
-- get info for variable 'name'
print("name=", table.tostring(sas.varinfo(dsid,"name")))
sas.close(dsid)
```

The table that **sas.varinfo** returns is identical in structure to the table that **sas.vars** returns.

PROC LUA provides a number of other functions to read data set metadata, and you can find out more about them in the PROC LUA documentation at http://support.sas.com/documentation/solutions/base/lua.

SUBSETTING DATA

When you read SAS data sets, it is sometimes necessary to restrict the set of observations that are read to observations that meet some criterion. You can subset data by using the **sas.where** function, as follows:

```
local dsid = sas.open("sashelp.class")
sas.where(dsid, "age>11")
while sas.next(dsid) do
   print(sas.get(dsid,"name"), sas.get(dsid,"age")
end
sas.close(dsid)
```

An active where clause can be augmented by using the ALSO keyword, rolled back by using the UNDO keyword, and cleared by using the CLEAR keyword, as shown in the following examples:

```
sas.where(dsid, "age>11")
     :
sas.where(dsid, "ALSO height<48")
     :
sas.where(dsid,"UNDO") -- undo the last where clause
     :
sas.where(dsid, "CLEAR") -- clear all active where clauses
```

WRITING TO SAS DATA SETS

The process of writing to a SAS data set involves first defining a new data set in terms of its name and structure and then opening it for updating and inserting observations. These tasks are described in the following sections.

DEFINING A NEW SAS DATA SET FROM SCRATCH

The **sas.new_table** function enables you to define a new (empty) data set. You provide the name of the new data set, and you provide a Lua table that is set up as an array of Lua tables, each of which contains the attributes of one data set variable. For example, the following code creates a new SAS data set called Work.Status that contains three variables, **node**, **status**, and **datetime**:

```
sas.new_table("work.status", {
           { name="node", type="C", length=36},
           { name="status", type="C", length=16},
           { name="datetime", type="N", length=8, format="DATETIME19."}
        })
```

Format, informat, length, and label specifiers are supported.

WRITING TO A NEW SAS DATA SET

After you define a new SAS data set, you can use it for output by opening it in update mode. You open it in update mode by specifying **u** as the second argument to the **sas.open** function, as in the following statements:

```
local dsid = sas.open("work.status", "u")
```

You then append observations to this data set by using the **sas.append** function.

A typical scenario is to write out observations in a loop of some sort, as follows:

```
local dsid = sas.open("work.status", "u")
for|while some-condition do
      :
      local node = ...
      local status = ...
      local dt = ...
      sas.append(dsid)
      sas.put(dsid, "node", node)
      sas.put(dsid, "status", status)
      sas.put(dsid, "datetime", dt)
      sas.update(dsid)
end
sas.close(dsid)
```

The **sas.append** function creates a new observation for the data set. The **sas.put** function places values in that new observation. Finally, the **sas.update** function is called to write the new observation with those values to the data set.

UPDATING AN EXISTING SAS DATA SET

You can update an existing data set by opening it in update mode. You locate the observation you want to update by using a where clause and then using the **sas.put** function, followed by **sas.update**, to save the new values into that observation, as shown in the following statements:

```
local dsid = sas.open("work.status","u")
sas.where(dsid, "node='segmentation'")
sas.put(dsid, 'status', 'completed')
sas.update(dsid)
sas.close(dsid)
```

ADDITIONAL FUNCTIONS

sas.exists and sas.fileexists

SAS functions treat 0 as false, whereas Lua treats 0 as true. Therefore, code like the following will not work as intended:

```
if sas.exist("work.foo") then
 :
```

The correct code would be:

```
if sas.exist("work.foo")~= 0 then  -- correct, but not intuitive
 :
```

The **sas.exists** function has been added to return a Boolean result to allow for more intuitive code:

```
if sas.exists("work.foo") then  -- correct
 :
```

The function **sas.fileexists** works in the same way, by returning a Boolean result as opposed to the 0 or 1 result that **sas.fileexist** returns.

sas.glibname and sas.gfilename

If you allocate a SAS library from Lua, either by calling the **sas.libname** function or by submitting SAS code (using **sas.submit**), you will find that the library allocation does not persist beyond the execution of PROC LUA. It does not persist because your Lua code executes within a PROC environment, and SAS operations such as allocating a SAS library or a new fileref limit their scope to their immediate environment.

If you need a SAS library to persist beyond the duration of the PROC LUA execution, use the **sas.glibname** function. Prefixing the function name with a **g** tells PROC LUA to target the global environment, which lives outside the PROC LUA boundary. For example, the following statements use the **sas.glibname** function to allocate a SAS library from Lua, and then make use of that library in a SAS DATA step that follows the PROC LUA statements:

```
proc lua;
submit;
 sas.glibname("MYDATA", "/home/fred/mydata")
endsubmit;
run;

data mydata.orders; /* use the library allocated in Lua */
 :
run;
```

WORKING WITH LARGER PROJECTS

Using embedded statements within the PROC LUA invocation works well for running ad hoc snippets of Lua code, but it has the following limitations:

- When programs are large, it becomes difficult to maintain all the code in one SAS program.
- Because the code is embedded in a SAS procedure invocation, you cannot use editing tools that support Lua syntax when you edit your Lua source code. So you lose the benefit of features such as syntax highlighting and context awareness.
- Because of the way the SAS language processor handles SUBMIT/ENDSUBMIT blocks, using embedded Lua code in PROC LUA does not work when that code is incorporated into a SAS macro.

Therefore, another mechanism is provided via PROC LUA's INFILE= option, which enables you to reference a Lua file by name (without the *.lua* extension). The following SAS statements reference the **hello.lua** file:

```
/* Referencing a Lua script called 'hello.lua'  */
proc lua infile='hello';
run;
```

Because there are no SUBMIT/ENDSUBMIT blocks in this invocation, this code can be wrapped inside a SAS macro. This approach is more appropriate when you build up a set of Lua programs that act as modules that can be called by each other. When you use this approach, you must place your Lua source file somewhere in the Lua search path so that PROC LUA can find it.

Lua SEARCH PATH

The Lua search path is similar to the SASAUTOS path, which is used to locate SAS macro.

You set up the search path by issuing a FILENAME statement that uses *LuaPath* for the fileref before you invoke PROC LUA, as shown in the following example:

```
filename LuaPath '<path-to-directory-containing-your-Lua-file>';
```

You can also use concatenated paths:

```
filename LuaPath ('<path1>' '<path2>' ...etc.);
```

For example, suppose you have the following Lua program:

```
print('Hello from Lua')
```

If you have saved this program as *c:\MyLuaFiles\hello.lua*, then you could run it from SAS as follows:

```
filename LuaPath 'c:\MyLuaFiles';
proc lua infile='hello';
run;
```

Note that you omit the *.lua* extension when you specify the INFILE= option.

When you use concatenated paths in your LuaPath FILENAME statements, PROC LUA loads the first file it finds in the search path whose name matches the one specified in the INFILE= option.

Lua MODULES

The value of the LuaPath fileref determines not only how Lua files are located by the INFILE= option of PROC LUA, but also how files are located when Lua processes a **require** statement, a mechanism that is used for loading Lua modules from within Lua. For example, the **split_date** function from the earlier example could be contained within a Lua script of its own and then reused by various Lua programs, as in the following example:

```
--
-- utils.lua
--
local module = {}
module.split_date = function(list)
  local date = sas.date() -- get current date
  return sas.day(date), sas.month(date), sas.year(date) -- return day, month, year
end
return module -- return table containing 'split_date' function to caller
```

The preceding code defines a local table variable called **module** and then inserts the **split_date** function in that table, using **split_date** as the key. The last statement returns the table to the caller. The table thus acts as a module, which is just a table of functions.

If you save this code as *utils.lua* under *c:\MyLuaFiles*, then other Lua scripts on that same search path can use make use of that function by using the following statements:

```
-- the table returned by 'utils.lua' is assigned to 'myUtils'
local myUtils = require('utils')
-- access the split_date function in the table
local day,month,year = split_date()
```

The **require** statement is very similar to the INFILE= option of PROC LUA. The main difference is that the INFILE= option always causes the specified Lua file to execute, whereas the **require** statement follows Lua's own rule of loading the code only if it has not been previously loaded. Thus the **require** statement is typically used to load *modules* (that is, Lua tables of functions and values) into the calling program, whereas the INFILE= option is used to call the main entry point into a Lua program.

The Lua search mechanism enables you to hierarchically structure your various Lua modules. For example, you could use your company's name to distinguish your site's *utils* module from a third-party version by storing *utils.lua* under a subdirectory of *c:\MyLuaFiles* (for example, *c:\MyLuaFiles\com\mycompany*). In that case, your **require** statement would become the following:

```
local utils = require('com.mycompany.utils')
```

The INFILE= option also supports this type of hierarchy.

LIMITATIONS

Currently, PROC LUA does not support all the modules that Lua provides. For example, PROC LUA does not currently support the **os** module, which offers direct support to the operating system. Lua's language features for multithreading have also been disabled.

The Lua community has created a large set of Lua modules that are available online. However, you might find that some third-party Lua modules do not work in PROC LUA. In particular, if they were written in C or involve custom-built C libraries, PROC LUA cannot run them. Lua code that is written for a Lua version other than 5.2 might need some modifications to work in PROC LUA.

COMPARISONS WITH THE SAS MACRO LANGUAGE

Traditionally, a series of DATA and PROC steps is controlled through the use of the SAS macro language. For large or more complex programs, or for projects that involve many other contributors, Lua's features can help make these projects more manageable. The following sections summarize some of the key differences between the SAS macro language and Lua.

DATA STRUCTURES

Because the SAS macro language is based on text, there is very limited support for data structures that can be represented in a SAS macro variable. Although a list of items can be implemented as a space-delimited series of strings, this can get cumbersome if the strings themselves contains spaces. Hash tables, trees, or other more complex structures would be much more difficult to implement. In contrast, Lua tables enable you to represent rich data structures in a single variable, and they are accompanied by a set of functions to help you manage those data structures.

ITERATION

The differences in iteration features between the SAS macro language and Lua can be best highlighted through a simple example.

Here is how you might use the SAS macro language to process a list of names:

```
/* Macro */
%let names=George Paul Ringo John;
%let n = 1;
%let nm = %scan(&names, &n, " ");
%do %while( &nm ne %str());
    %put &name;
    %let n = %eval(&n + 1);
    %let nm = %scan(&names, &n, " ");
%end;
```

Compare this with Lua:

```
local names={"George", "Paul", "Ringo", "John"}
for i,v in ipairs(names) do
     print(v)
end
```

By eliminating the need for the various `%` and `&` characters, the code becomes more readable and easier to maintain and debug.

FUNCTIONS

The SAS macro language offers support for returning values from macros, but in some cases modularity is broken because the internals of how a macro is written determines how the macro can be invoked.

For example, if your macro does not include any DATA or PROC steps, you can return a value in this manner:

```
%macro do_something;
 %local somevar;
  :
 &somevar   /* return the value *?
%mend

%let a=%do_something;
```

If you were to introduce a PROC step into that macro, you would no longer be able to use that macro as originally intended. The following code fails:

```
%macro do_something;
proc sql;
  select max(age) into :foo from sashelp.class;
 quit;
 &foo
%mend;
%let a=%do_something; /* generates syntax error */
```

Thus, if you adopted this return style when you first implemented the macro and then later you needed to introduce a SAS PROC or DATA step into it, you would end up needing to modify all the calls made to this macro, because you would now have to call this macro differently.

You might solve this problem by using code like the following:

```
%macro do_something(in=,out=);
  %local somevar;
  :
  %let &out=Hello;  /* hope that caller didn't use 'somevar'
                       as the name of the output variable */
%mend;


%local a;

%do_something(in=foo, out=a);
%put a=&a;
```

However, this approach suffers from another limitation: a potential name conflict between the OUT= parameter value and any local variables that are used by that macro. In Lua, issues like these do not exist because functions offer true encapsulation, as shown in the following statements:

```
function do_something(in)   -- caller does not need to
   local variable           -- to know the local variable
   local somevar            -- names of this function
    :
   return "Hello"
end

local v1 = do_something("foo")
```

Returning multiple values is even more challenging in a macro, forcing you to either accept greater proliferation of global variables or use a similar OUT= parameter as in the previous example. Either alternative forces the caller of the macro to become familiar with the internals of that macro, in order to know what names not to use for any OUT= parameters so that they do not conflict with any of the macro's local variables.

Lua can return multiple values in two ways. One way is by putting all of the return values into a table and returning the table, as in the following example:

```
function do_something(in)
  local somevar
   :
  return {color="red", make="Honda", price=12500}
end

local v1 = do_something("foo")
-- v1= {color="red", make="Honda", price=12500}
```

As seen earlier in the section "FUNCTIONS" on page 4, the other way Lua can return multiple values is to return them as a comma-separated list, which can then be received by a comma-separated list of variables.

Calling SAS functions is straightforward in Lua. You use the **sas.** prefix instead of wrapping the function call in the %SYSFUNC macro. For example, in the SAS macro you would use the following statement to invoke the RAND function:

```
%let x = %sysfunc(rand(Lognormal))
```

In Lua, you would just use the following statement:

```
x = sas.rand("Lognormal")
```

ERROR REPORTING

Lua syntax errors are reported with the line number of the offending Lua statement. For example, the following error message indicates an error in line 9 of the Lua code:

```
ERROR: There was an error loading the file
ERROR: test.lua:9: unexpected symbol near ')'
```

Contrast this to errors that are reported by the SAS macro processor, such as the following:

```
ERROR: An unexpected semicolon occurred in the %DO statement.  A dummy macro
       will be compiled.
```

This is also true of run-time errors—Lua reports the line number of the location where the error is first encountered.

Lua offers this benefit because it does not use the SAS macro preprocessor approach to code execution. You will experience this benefit when you when debug your programs, especially for large programs that involve hundreds or thousands of lines of code. The result of macro expansion is a set of SAS statements that might not correspond, line-by-line, to the statements in your original source file. Thus, when errors that are reported in the SAS log refer to line numbers, those line numbers are often not useful to you. On the other hand, Lua reports back line numbers that you can relate to the lines in your source file—so you can quickly locate the problem area and address the issue.

ARITHMETIC

In the SAS macro language, it is necessary to use the %EVAL macro to indicate where a text expression should be interpreted numerically. In Lua, this is not necessary because numeric data types are supported directly. There is also no need in Lua to distinguish between floating point and integer operations, as is the case in the SAS macro language, where you need to know when to use %EVAL and %SYSEVALF. Thus, to increment a counter variable in the SAS macro language, you need to write the following:

```
%let count = %eval(&count + 1)
```

whereas in Lua, you can simply write the following:

```
count = count + 1
```

CONCLUSION

The LUA procedure offers a new approach to writing SAS code by enabling you to use Lua to orchestrate your PROC and DATA steps. This difference is especially noticeable when you work with larger and more complex programs that can be very difficult to implement using only the SAS macro language for the control logic. The fact that PROC LUA and the SAS macro language can work together lets you decide how much of one approach versus the other to adopt.

REFERENCES

Ierusalimschy, R. (2013). *Programming in Lua*. 3rd ed. Rio de Janeiro: Lua.org.

Lua.org (2013). "The Programming Language Lua." http://www.lua.org.

ACKNOWLEDGMENTS

The author is grateful to Donald Erdman for his guidance in the creation of PROC LUA, for some code examples used in this paper, and for reviewing this paper. Rick Langston's assistance in preparing PROC LUA for release and in reviewing this paper has been immeasurable. Dan Jackson provided valuable input into the design of the auxiliary functions and string handling routines. Tim Hunter prepared much of the ground work for embedding Lua in SAS. The author also thanks Anne Baxter for editorial assistance.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Paul Tomas
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-5339
Paul.Tomas@sas.com