Paper 212-2017

A Comparison of the LUA Procedure and the SAS® Macro Facility

Anand Vijayaraghavan, SAS Institute Inc.

ABSTRACT

The LUA procedure is a relatively new SAS® procedure, having been available since SAS® 9.4. It enables you to use the Lua language as an interface to SAS and as an alternative scripting language to the SAS macro facility. This paper compares and contrasts PROC LUA with the SAS macro facility, showing examples of approaches and highlighting the advantages and disadvantages of each.

INTRODUCTION

Lua was developed by Roberto Ierusalimschy at the Pontifical Catholic University in Brazil. It is distributed as open-source software and is documented both online and in a number of books, some of which are cited as references at the end of this paper. Lua is a popular choice in game development and many prominent games such as World of Warcraft and Angry Birds, which are video games that are scripted in Lua.

The Lua 5.2 run-time environment is embedded within SAS and makes SAS functionality available through Lua. Conversely, Lua 5.2 also provides for Lua code to be run from SAS using PROC LUA. PROC LUA makes Lua available as a general-purpose scripting language within the SAS language, which is suitable for building large-scale modular solutions.

The first section of this paper elaborates on the choice of Lua as the scripting language and the motivation behind comparing it with the SAS macro facility. The second section provides examples of situations where PROC LUA is more appropriate than the SAS macro facility. The third section highlights some interesting differences between Lua and the SAS® Macro Language. The fourth section elaborates with example situations where the SAS Macro Language is more appropriate than PROC LUA. The fifth section provides some similarities between the SAS Macro Language and the Lua programming language. Finally, the conclusion section summarizes the discussion of this paper.

WHY LUA?

The need for an alternate to the SAS Macro Language was first felt by various solutions group within SAS. These teams had moved beyond the ability of the SAS Macro Language to support productive development of sophisticated SAS solutions. It was therefore felt that there was a strong need for a modern scripting language in addition to the SAS platform. Since the purpose of Lua is to script C-based software and because SAS is written in C, Lua was a great fit for SAS. Several other teams have since evaluated scripting options and have had success using Lua with SAS.

Internal discussions at SAS resulted in the following observations:

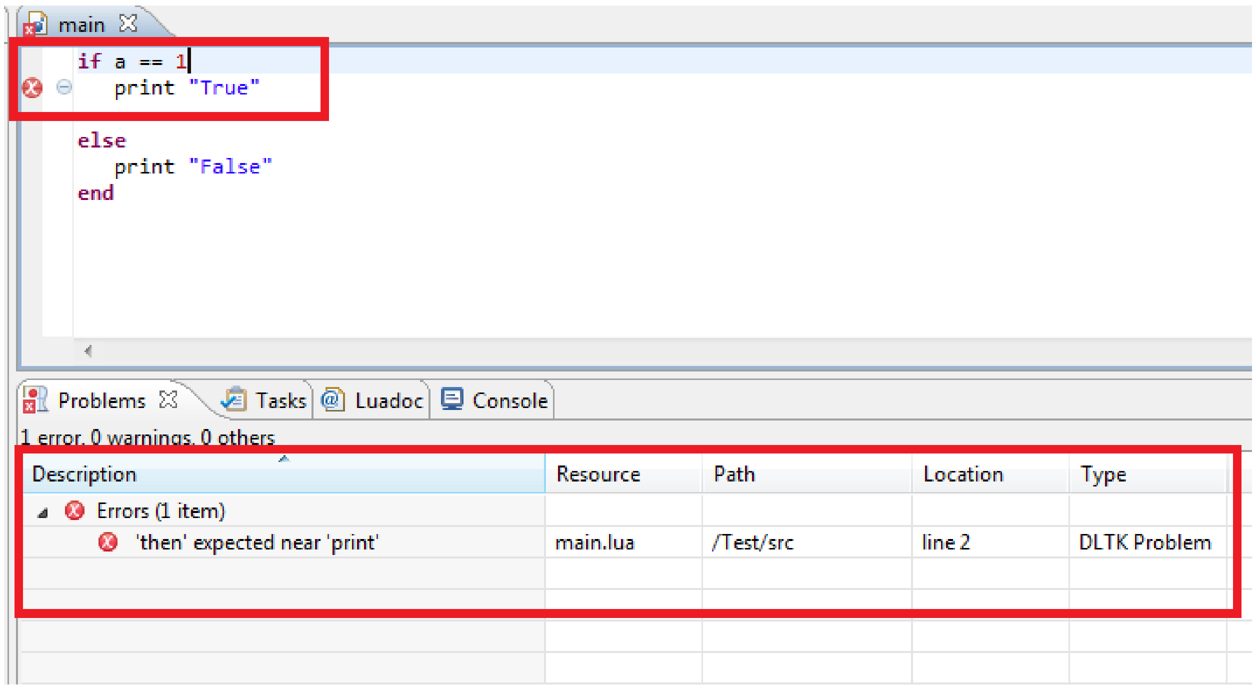
1. SAS solutions development based on the SAS macro facility had long been held back by a lack of modern language structure and development tools.
2. Lua provides for Eclipse plug-ins and provides more detailed debugging information, which the SAS Macro Language lacks. It also would be technically infeasible to enhance the SAS macro facility to equal Lua's structure and ease of debugging.
3. Lua has been widely adopted in the industry, and the modern language features of Lua will resonate with users who are new to SAS but familiar with popular scripting languages.
4. Lua is a fast, portable robust language, which is open source. Furthermore, it was designed to script C-based applications and is blank insensitive, which makes it a good fit for SAS.

SITUATIONS WHERE LUA IS A GOOD CHOICE

While the SAS macro facility is a powerful tool to have in the SAS arsenal, it has its limitations. The SAS macro facility was originally intended to be a text generator. Over the years, users have sought to and succeeded in using it for far more than this. Many of the enhancements that have been made over the past 35 years have, to a great degree, contributed to users being able to do so. However, the SAS macro facility inherently limits what it can do. It is in these situations that the functionality of the Lua programming language via PROC LUA proves to be a very valuable tool for SAS users. Furthermore, when you are working on large projects to develop solutions, the practical ease of coding and code maintenance becomes crucial. Listed below are some cases where Lua is the better choice.

SYNTAX HIGHLIGHTING

Text editors and IDEs such as Eclipse recognize Lua syntax and support syntax error highlighting. Display 1 shows an example of a situation where a small syntax error can be easily caught early during development:



```
if a == 1
   print "True"

else
   print "False"
end
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
'then' expected near 'print'	main.lua	/Test/src	line 2	DLTK Problem

Display 1. Eclipse IDE Syntax and Error Highlighting

IDEs are important because they provide an interface for developers to compile and execute code incrementally. They also enable you to manage changes to source code in a uniform manner. The SAS macro facility lacks such integration with IDEs. The code provided above is just a small example to illustrate the point. With thousands of lines of code, it would be very time consuming to compile and then debug a large piece of code and then realize that there is a syntax error somewhere in the middle.

FUNCTION TRACEBACK

Error reporting and terminations on error conditions are cleaner in Lua and provide more detailed debugging information. This is especially useful in large projects where pinpointing the exact line can save a lot of time and effort in fixing issues.

Consider the following Macro code where there is a misplaced semicolon:

```
%macro abc;
   %let a=;Hello;
   %put &a;
%mend abc;
%macro xyz;
   %abc
%mend xyz;
%macro pqr;
   %xyz
%mend pqr;
%pqr
```

Resulted Log:

```
NOTE: Line generated by the invoked macro "ABC".
             Hello;

             _____
             180
ERROR 180-322: Statement is not valid or it is used out of proper order.
```

There is no clear indication of which line caused the error. The SAS macro facility is limited in this sense because it follows a preprocessor approach. In that approach, each macro expands into a series of SAS statements and therefore can't point to the original source line. In comparison, consider the example below where we have a Lua script test.lua, which is invoked by a SAS program:

Test.lua:

```
function abc()
      assert(1 > 3,"This is an error message")
end
function xyz()
      abc()
end
function pqr()
      xyz()
end
pqr()
```

SAS Code:

```
filename LUAPATH "";
proc lua infile='test';
run;
```

Resulting Log:

```
ERROR: ./test.lua:2: This is an error message
stack traceback:
    [C]: in function 'assert'
    ./test.lua:2: in function 'abc'
    ./test.lua:6: in function 'xyz'
    ./test.lua:10: in function 'pqr'
    ./test.lua:13: in main chunk
    [C]: in function 'require'
```

The traceback lists the entire sequence of calls made along with the filename and the line numbers in the Lua code before the error condition is hit. It is clear that the second line in the code inside the function abc() in the file test.lua is the cause and thus can be easily fixed.

MULTIPLE RETURN VALUES

Lua functions can return multiple values as results. This feature is especially useful when routines perform more than one task and thus would automatically report all the results. Consider the example below:

```
proc lua;
submit;
   function arith(num1, num2)
    local sum = num1 + num2
    local prod = num1 * num2
    return sum, prod
   end
   sum, prod = arith(10, 20)
endsubmit;
run;
```

Now here is the same code in the SAS Macro Language:

```
%macro arith(num1,num2,sum=,prod=);
   %let &sum = %eval(&num1 + &num2);
   %let &prod = %eval(&num1 * &num2);
%mend arith;
```

```
%global out1;
%global out2;
%arith(10,20,sum=out1,prod=out2)
```

Although this might at first seem like a good solution, it has its drawbacks. When the macro is invoked, the calling code should have the information about the variables **`sum`** and **`prod`**. These are internal to the macro definition, but need to be known outside. Furthermore, these variables have now become reserved in a way. Thus, local variable names need to be different from **`sum`** and **`prod`** to avoid conflict.

In summary, although returning multiple values is feasible in the SAS Macro Language, it is not a clean solution, and the possibility of an error is high when working on large projects.

VALUE TYPE

Everything in the SAS macro facility is seen as text. Lua does not have a concept of a data type. But it does provide for eight value types. As the name suggests, the type is associated with the value itself rather than the variable. This is because a variable can always be overwritten with a value of a new type. The table below lists the Lua types available:

Type	Meaning	Example
nil	Represents absence of a value	Nil (single value only)
Boolean	True or False	True / False
number	Double precision floating point	1.2356
string	Sequence of characters	“Hello World”
table	Associative array	x[“key”] = 100, x[2] = “value”
function	Represents a method	type(), print()
userdata	Arbitrary C data	lua_newuserdata() used
thread	Represents Threads of execution	Coroutine

Table 1. Lua Value Types

The table and function types are especially useful. The function type is used to store a function in a variable and can thus be used to pass a function as an argument to another routine or even return a function as a result. The table type acts as a hash table, which is a very powerful data structure, which has many applications when developing solutions.

ITERATORS

Using iterators in Lua is considerably easier than in the SAS Macro Language. The simplicity of the syntax and different custom iterators available make it a more powerful tool, which can be used in a variety of situations. Consider the example below:

```
%macro iterator;
   %local dsid;
   %let dsid = %sysfunc(open(sashelp.class));
   %do %while(not %sysfunc(fetch(&dsid))) ;
       %let age=%sysfunc(getvarn(&dsid,%sysfunc(varnum(&dsid, age))));
       %let height=%sysfunc(getvarn(&dsid,%sysfunc(varnum(&dsid, height))));
       %put &age &height;
   %end;
   %let dsid = %sysfunc(close(&dsid));
%mend iterator;
%iterator
```

The above code in the SAS Macro Language reads a data set and then prints the age and height from each of the observations. Such a simple task requires many macro calls and seems complex, so it is not easily understood at first glance. Here is the same code, now using PROC LUA:

```
proc lua;
submit;
   local dsid = sas.open("sashelp.class")
   for row in dsid:rows() do
      print(row.age, row.height)
   end
   dsid:close()
endsubmit;
run;
```

Lua offers a much cleaner way of coding. Lua also has access to stateless iterators such as the key/value pair iterators, which can be used to traverse hash tables:

```
proc lua;
submit;
   x = {day="Monday", month="Feb", year=2017}
   for i,v in pairs(x) do
      print(i, v)
   end
endsubmit;
run;
```

INTERESTING DIFFERENCES BETWEEN LUA AND SAS MACRO

The following examples highlight some common mistakes that a Lua user might make while using the SAS Macro Language. The examples also highlight how the interaction of PROC LUA and MACRO might produce interesting results.

USE OF SEMICOLON

The Lua syntax does not mandate that every valid statement end with a semicolon. The user is free to do so, but it is optional. For example, the following two statements are both valid:

```
local a = “Hello”
local b = “World”;
```

Lua goes one step further and does not even mandate a semicolon when there are multiple statements on the same line. But in such a situation, it is advisable to have one to avoid ambiguity as shown below:

```
local a = “Hello” local b = “World”       -- Ambiguous without semicolon
local a = “Hello”; local b = “World”;
```

In contrast, the SAS Macro Language follows the SAS statement syntax in the sense that every valid statement must end with a semicolon. A long-time Lua user might be in the habit of forgetting to do so, which might lead to additional efforts in debugging later.

SUBMITTING LARGE MACROS INSIDE PROC LUA

The sas.submit() function in PROC LUA is used to submit SAS code from the Lua environment. But it runs into interesting problems when coupled with submitting large SAS Macros. Consider the example below:

```
proc lua;
submit;
    sas.submit
    ([[
          %macro abc;
          …
            -- Large Macro
            …
          %mend abc;
          %abc
    ]])
```

```
endsubmit;
run;
```

Resulting Log:

```
ERROR: A dummy macro will be compiled.
WARNING: Missing %MEND statement for macro ABC.
```

The above error and warning messages are a result of the way that the sas.submit() function works. All code within the sas.submit block is submitted in batches to SAS. Typically, the code is sufficiently small for it to be submitted in a single batch. But, when the macro inside this block is large—say larger than 32k—then there is a buffer overflow, and the code needs to be submitted in multiple batches. The SAS macro facility cannot handle batches such as these. It expects all code that is part of a single macro to be submitted as one single block. Thus, when the first batch of code is submitted, the SAS macro facility parses the code and searches for a %mend, which it will not find because it is a part of a subsequent batch and will thus put out the error and warning messages.

The workaround for this is to use the alternate sas.submit_() function, which makes sure that we do not break down a macro. This function will queue up the code, but doesn't actually perform the submit that is required. A subsequent sas.submit() function call would then be used to submit all the code:

```
proc lua;
submit;
    sas.submit_
    ([[
           %macro abc;
           …
             -- Large Macro
             …
           %mend abc;
    ]])
    sas.submit([[%abc]])
endsubmit;
run;
```

GLOBAL VERSUS LOCAL VARIABLES

All variables in Lua default to a global scope unless they are explicitly quantified otherwise using the local keyword. Consider the example below:

```
proc lua;
submit;
   num = 10             -- global
   if num > 5 then
       local num      -- local to "then" block
       num = 20
       x = 100        -- defaults to global
   end
   print(num)           -- 10 (global)
   print(x)             -- 100
endsubmit;
run;
```

If the num declared inside the "if" block has not been explicitly declared as local, then its value would be overwritten, and we would have a different result. Furthermore, the variable "x," which is first used here inside the "if" block, will default to a global scope and will thus exist even outside its scope. So it is the responsibility of the user to keep track of all such variables. In contrast, consider the same code written in macro:

```
%macro abc;
    %let num = 10;
    %if %eval(&num > 5) %then %do;
           %local num;                /* local to macro abc NOT %do block */
           %let num = 20;
           %let x = 100;
    %end;
    %put &num;                       /* num = 20 */
    %put &x;                         /* x = 100 */
%mend abc;
%abc
```

The %local and %global have different meanings in the SAS Macro Language as compared to Lua. The local scope in the SAS Macro Language refers to the locality of the scope within a particular macro, and it does not refer to a particular block (the %do block in this case). Thus, we see that variable **`num`** is local to the entire macro abc and not the %do block. Its value is affected by the %let statement, and this change persists outside the block as well.

SITUATIONS WHERE SAS MACRO LANGUAGE IS A BETTER CHOICE

FAMILIARITY WITH SAS MACRO LANGUAGE

The SAS Macro Language was first introduced in SAS® 82 and has been an integral part of SAS programming over the past 35 years. Over these years, SAS customers have gained familiarity and have become comfortable coding with the Macro Language. In comparison, Lua is a newer language and has gained popularity only in recent years. Although Lua is used in a variety of fields and for a variety of applications, the majority of these users are from the video game development industry, which narrows down the user audience. Therefore, a randomly selected SAS customer would definitely have heard of and would have used SAS Macro, but the same cannot be said for Lua, which is in itself a big drawback.

INTEGRATION WITH SAS TOKENIZER

The SAS Macro Language is part of the SAS macro facility, which is part of the Base SAS® product. It gives the SAS programmer the ability to programmatically manipulate SAS source code through symbolic substitution. This flexibility is a direct result of the fact that the SAS Macro Language is recognized by and integrated with the SAS language tokenizer. The trigger characters, “&” and “%,” are used to signal to the tokenizer that the statements belong to the SAS Macro Language and need to be handled accordingly. This ensures that SAS Macro Language code can be interwoven into the SAS code, providing seamless integration of the two. Consider the example below where both SAS and Macro code are used:

```
%let num = 10;                     /* Macro code in open code */
data _null_;
   put "%sysfunc(today(),worddate)";/* Combination of SAS and Macro Code */
run;
```

In contrast to this, Lua is not integrated with the SAS tokenizer. Therefore, Lua code cannot be used in open code and neither can it be used interchangeably with SAS code. The use of Lua is restricted to PROC LUA and cannot be used outside the scope of the procedure. This is probably the biggest drawback with PROC LUA for a SAS user.

CASE INSENSITIVITY

A major advantage with the SAS Macro Language is that the syntax is case insensitive. Both uppercase and lowercase keywords can be used alternatingly. Consider the example below:

```
%macro abc;
    %let a = Hello World;
    %put &a;
%mend abc;
%abc
```

The same can be rewritten in uppercase as follows:

```
%MACRO abc;
    %LET a = Hello World;
    %PUT &a;
```

```
%MEND abc;
%abc
```

PROC LUA does not support this feature because it is limited by the Lua parser syntax:

```
2          proc lua;
3          submit;
4                   LOCAL a = "Hello World"  -- Uppercasing will produce an error
5                   PRINT(a)
6          endsubmit;
7          run;
NOTE: Lua initialized.
SUBMIT block:1: syntax error near 'a'
ERROR: There was an error submitting the provided code
```

SIMILARITIES BETWEEN LUA AND THE SAS MACRO LANGUAGE

Despite all the differences and the comparisons of various situations in which one language outperforms the other, there are some key similarities between the Lua and the SAS Macro Language. These similarities would especially be useful in projects and situations that involve both languages.

CALLBACKS TO SAS

PROC LUA enables users to make callbacks to SAS functions from a Lua environment. This is achieved with the help of a global Lua table called “sas”. Thus, all SAS functions can be called using the prefix “sas”. Consider the example below which makes a call to the SAS exist() function to check if a given data set exists or not:

```
proc lua;
submit;
    if(sas.exist("work.homes") == 1) then
          print "Data Set exists"
    else
          print "Data Set does not exist"
    end
endsubmit;
run;
```

The same functionality can be achieved using the SAS Macro Language. The %sysfunc macro function is used in such cases to make calls to SAS functions:

```
%macro check;
    %if %sysfunc(exist(work.homes)) = 1 %then %do ;
          %put Data Set exists;
    %end;
    %else %do;
          %put Data Set does not exist;
    %end;
%mend check;
%check
```

SAVE COMPILED CODE

The SAS macro facility compiles and saves macros in a permanent catalog in a library that is specified by the user. The advantage with this approach is that the compilation occurs only once. If the stored compiled macro is called in the current or later sessions, the macro processor executes the compiled code. The following example shows how this works:

```
libname mypath ‘C:\MACROS’;
options MSTORED SASMSTORE=mypath;
%macro dummy /store;
    %put This macro will be compiled and stored in a catalog;
%mend dummy;
```

The above code will compile and store the Macro %dummy in the catalog called SASMACR in the SAS library 'mypath'. The following syntax can be used to invoke the macro in a subsequent SAS session:

```
libname mypath 'C:\MACROS';
options MSTORED SASMSTORE=mypath;
%dummy
```

Lua code can also be compiled and saved as a LUC file. PROC LUA enables users to later call the compiled code using the INFILE option. Consider the following example where a piece of Lua code was compiled and stored as **`mytest.luc`**.

```
proc lua infile='mytest';
run;
```

CONCLUSION

PROC LUA combines the advantages of using a powerful scripting language Lua, along with all the functionality of SAS. It provides for a framework in which SAS functions can be called from Lua. The integration of Lua with SAS does not preclude the use of both Lua and SAS Macro Language together. Lua can be used to submit SAS code that uses macros. Thus, you can take advantage of both languages and grow the use of Lua in macro code. The author of this paper does not say that one language is better than the other. Based on a few examples, this paper aims only to highlight the advantages of using each of them. Depending on the requirements and specifications of the project at hand, a choice for one or the other or even a combination of both can be used.

REFERENCES

Ierusalimschy, R. (2013). *Programming in Lua*. 3rd ed. Publisher: Author.

Ierusalimschy, R., L. H. De Figueiredo, and W. Celes. 2006. *Lua 5.1 Reference Manual*. Lua.org.

Lua Official website: https://www.lua.org.

Tomas, P. 2015. "Driving SAS with Lua." Proceedings of the SAS Global Forum 2015. Cary, NC: SAS Institute Inc. Paper SAS 1561-2015, available at http://support.sas.com/resources/papers/proceedings15/SAS1561-2015.pdf.

ACKNOWLEDGMENTS

The author is grateful to Rick Langston and Paul Tomas for their mentoring with the SAS Macro Language and PROC LUA respectively and for providing some of the code examples used in this paper. Donald Erdman's valuable input regarding the challenges faced by the RISK team and his inputs on the advantages of PROC LUA has been crucial to this paper. The author is also thankful to Rick, Paul, Donald, Patrick Garrett, and Russ Tyndall for their feedback and help in reviewing this paper.

CONTACT INFORMATION

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

Anand Vijayaraghavan
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 531-3428
Anand.Vijayaraghavan@sas.com