Paper 3824-2019

Automating Migration from the SAS® Macro Language to the LUA Procedure Using Transpiling

Igor Khorlo, Syneos Health™

ABSTRACT

Many solutions based on SAS® use the SAS macro language to structure, automate, or make SAS code more dynamic. When you scale out to a large application with thousands of lines of macro code, SAS macro usually becomes a bottleneck in the development and starts to hit its limitations. Maintenance, debugging, and testing can become a challenging and time-consuming process even for experienced SAS programmers. The Lua language was designed to be embedded into applications to provide scripting functionality. Compared to alternatives like PROC GROOVY that runs in a separate JVM process, PROC LUA (introduced in SAS® 9.4M3) runs inside a SAS process and, as a result, has access to the C function bindings that SAS has for reading, writing data sets, and so on. With that plus its performance, small footprint, elegant syntax, and support for data structures, the PROC LUA became a very promising replacement for a SAS macro for big projects.

A transpiler is a program that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language. In other words, it is a source-to-source compiler (like PROC DSTODS2 in SAS).

In this paper, we consider performance differences in terms of CPU, memory, and disk I/O between PROC LUA and SAS macro for common coding situations, and build a transpiler[1] that parses SAS macro code, creates an Abstract Syntax Tree representation and translates it to Lua source code.

INTRODUCTION

SAS introduced the LUA Procedure[2] in SAS version 9.4 Maintenance 3. A good introduction into the Lua language in the scope of SAS language was made by Paul Tomas[3], an excellent introductory workshop was held by Rowland Hale on PharmaSUG 2018 China[4]. People from the SAS community continuously show interested in Lua language. The LUA procedure was evolving from the time it was introduced, and many uncertainties were addressed, the documentation getting better and now contain many useful examples, the os module was included, despite, it was initially disabled. The simplicity of the language, low entry threshold, its performance, and integrability into SAS environment makes it a promising replacement for the old fashioned SAS Macro.

PERFORMANCE

The main difference between SAS macro and Lua which improves efficiency so dramatically is that Lua holds all its variables/objects in memory while SAS macro needs to maintain views

[1] demo – https://saslint.com/sasmacro2lua
[2] SAS (2018)
[3] See Tomas (2015)
[4] Hale (2018a); see Hale (2018b) for the workshop materials

like SASHELP.VMACRO. This overhead results in a lot of disk I/O operations. The disk resource is usually a bottleneck in the performance. A simple loop like this:

```
%macro dummy_loop;
  %do i=1 %to 1000000;
    %put &i;
  %end;
%mend;
%dummy_loop;
```

can spawn utility files up to 100mb in the WORK library. That is an incredible waste of system resources for such a simple task:

```
$ ls -al
-rw-r--r--   1 sasdemo      sasdemo    88039424 Oct 20 19:57 #tf0024.sas7butl
drwx------   2 sasdemo      sasdemo        1024 Oct 20 19:57 .
drwx------   3 sasdemo      sasdemo          96 Oct 20 19:56 ..
-rw-r--r--   1 sasdemo      sasdemo       12288 Oct 20 19:57 sasmac1.sas7bcat
```

SAS utility file "#tf0024.sas7butl" has a size of 88039424 bytes which is 88 megabytes!

We are going to compare the performance for SAS macro and PROC LUA for the following coding situations that commonly arise in SAS macro development:

- loops
- iterating over a list
- reading a data set.

Performance of the SAS code can be monitored using SAS® 9.4 Interface to Application Response Measurement (ARM) which will output a bunch of tables with numbers for CPU, memory, disk, I/O, etc. Consequently, you would need to do a lot of data preparation and visualisation until you get something presentable and clear. We are going to use the Enterprise Session Monitor (ESM)[5] for the performance monitoring that will output performance profiling graphs via an intuitive, user-friendly web-based interface.

DO-loop (for loop)

First code example we are going compare is a simple DO-loop (for loop):

```lua
for i = 1, n do
  print(i)
end
```

Listing 1: Lua

```
%do i = 1 %to &n;
  %put &i;
%end;
```

Listing 2: SAS Macro

From here it is already noticeable how Lua syntax is different from SAS macro syntax – no redundant "%", no semicolons. We are going to compare the above examples for 10 million iterations (n=1e7). We run both examples one by one – first Lua one, second SAS macro after a little pause.

[5]Enterprise Session Monitor (ESM) for SAS is visual performance monitoring software by Boemska https://boemskats.com/esm/

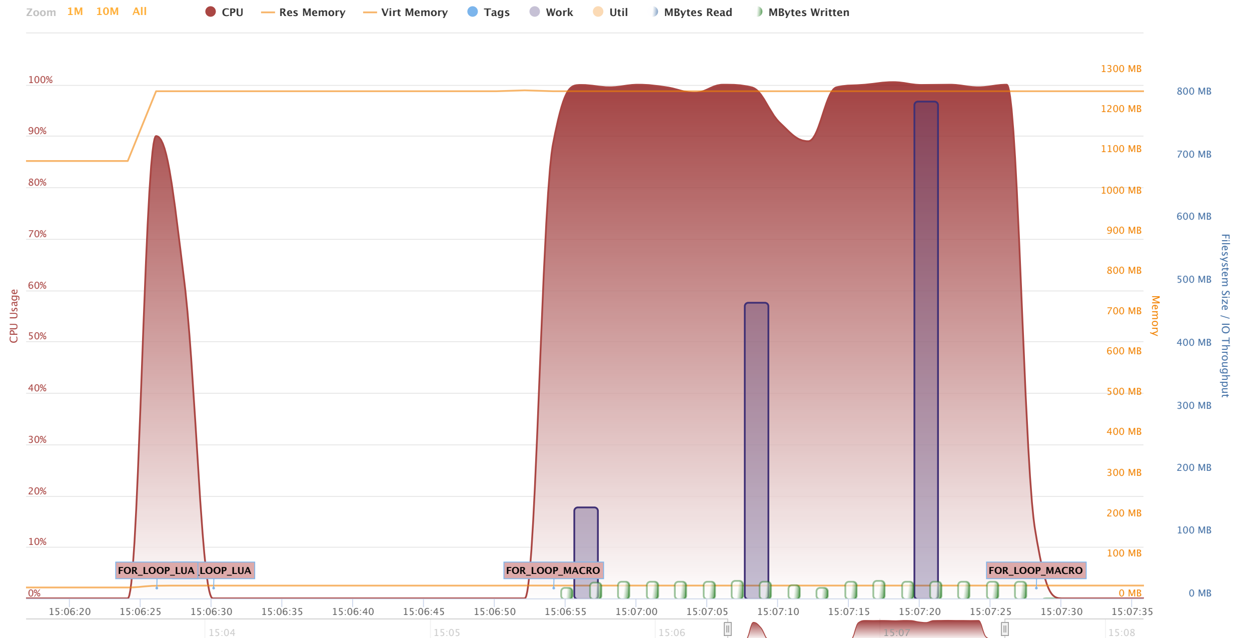

Figure 1: DO-loop comparison – PROC LUA on the left, SAS macro on the right. Red curve – **CPU**, yellow curve – **memory**, wide purple blue bars – **WORK directory size**, green bars – **I/O write**.

From the first look, it is already clear that the Lua version is faster and consumes fewer system resources. Before we consider this in more detail, I want to add a few notes about execution time measurement. There are boxes with labels FOR_LOOP_LUA, FOR_LOOP_MACRO that indicate bounds of each comparison unit. These boxes created out of SAS code using a tagging functionality in ESM that allows passing information from SAS session to the monitoring graphs. We are going to measure execution time and pass that value to ESM so that it will be visible to us at a closing tag.

- **Execution time.** Lua version took 3.03 seconds, which is 10 times faster than SAS macro, it took 34.35 seconds.
- **Disk.** Lua version had no impact on the WORK directory size, Lua did not create any utility files which is expected – it holds everything in memory. On the other hand, SAS macro version did create utility files in the WORK directory, and the size of the WORK directory increased from 0 to 793 Mb.
- **CPU.** Both versions were keeping CPU at ~100% load which was expected. However, the SAS macro version took 10 longer than Lua, which means that cumulatively it consumed more CPU resources.
- **Memory.** SAS macro did not show any noticeable memory use. For the Lua version, the situation is a bit different. In the beginning, memory use increased by 180 Mb – this is the memory footprint of an initialised Lua session. Amount of used memory did not change during execution. To free this memory up PROC LUA has TERMINATE= option. Although, 180 Mb is a tiny memory amount by modern standards.

The guesses were confirmed and as we can see, SAS macro creates utility files under WORK library of insane sizes. Let's see the next example.

Iterating over a list

The irony here is that SAS macro does not even have a built-in list data structure, so a standard solution to that is to use a macro variable as storage where list elements are joined with a character, not present in any element. Elements can be retrieved afterwards using %SCAN Macro Function. Lua has an array-like data structure encapsulated into a basic type – table.

For simplicity, we are going to use a list of 9 words, that we are going to iterate 100000 times (n=1e5). The equivalent code in SAS macro and Lua will look like:

```lua
list = {"George", "Paul", "Ringo",
        "John", "Foo", "Bar",
        "Baz", "Macro", "Polo"}

for _, i in ipairs(list) do
  print(i)
end
```

Listing 3: Lua

```sas
%let list=George Paul Ringo John
          Foo Bar Baz Macro Polo;
%do i = 1 %to
  %sysfunc(countw(&list));
  %let item=%scan(&list, &i);
  %put &item;
%end;
```

Listing 4: SAS Macro

`ipair()` function in Lua returns a pair – `index`, `item`. However, as we don't need an index here, we just use a placeholder "_" for it – that is a conventional way to indicate a non-used variable.

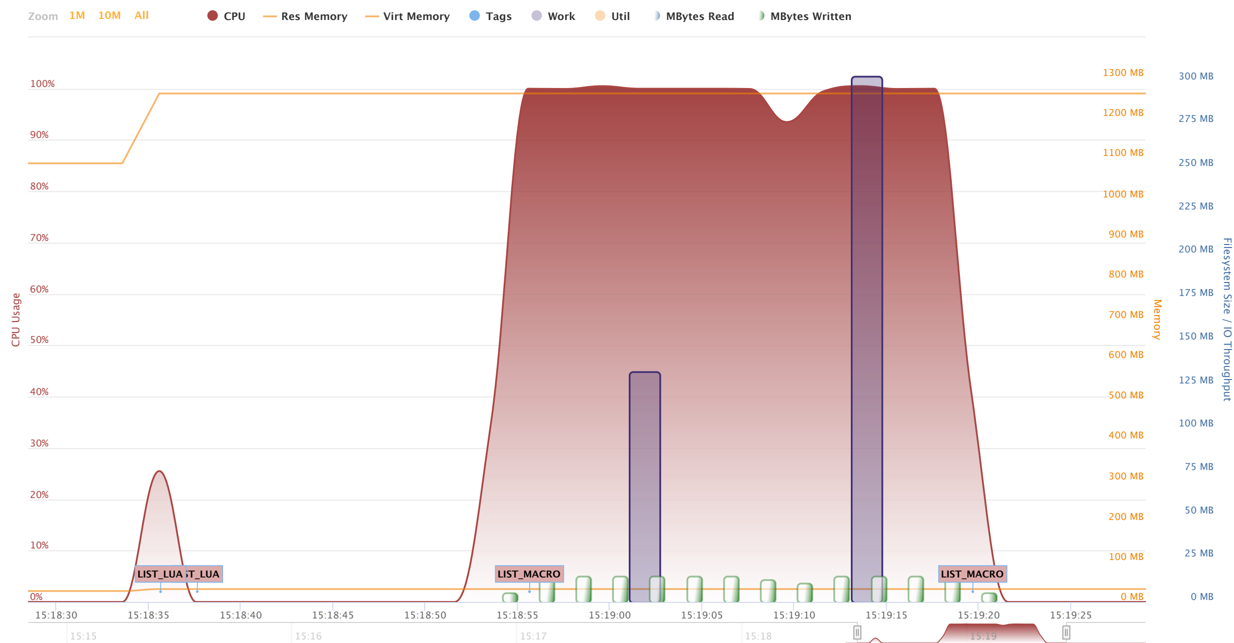

Figure 2: Iterating over a list comparison – PROC LUA on the left, SAS macro on the right.

From the graph, we can see the same performance patterns. For the Lua version that is even cannot be called a load. Let's consider each system resource:

- **Execution time.** Lua version took 777 milliseconds. For SAS macro is took 25.55 seconds.
- **Disk.** Same problem here for SAS macro – it is intensively writing to WORK directory. Lua version had no impact on disk.

- **CPU.** Lua version showed CPU usage <30% that could be barely called as a warmup, while SAS macro was burning 100% CPU for 25 seconds.
- **Memory.** SAS macro does not require memory; Lua needs around 180 Mb to initialise and hold its state.

Reading a data set

Another ubiquitous task arises in SAS macro development is to read a SAS dataset in a pure macro way. For testing purposes, we created an extended `SASHELP.CLASS` dataset in the following way:

```
data class(drop=i);
  set sashelp.class;
  do i = 1 to 100000;
    output;
  end;
run;
```

The resulting `CLASS` dataset has 1.9 million observations, 5 variables and is 73 Mb of size. The equivalent code in SAS macro and Lua will look like:

```
dsid = sas.open('class')

-- there are at least 3 ways
-- of reading a dataset in Lua
for obs in sas.rows(dsid) do
  -- now you can use obs.name
  --                 or obs['age']

  print(obs.name .. ', ' ..
        obs['age'] .. ' years')
end

sas.close(dsid)
```

Listing 5: Lua

```
%let dsid = %sysfunc(open(class));
%syscall set(dsid);
%let nobs =
  %sysfunc(attrn(&dsid, nlobs));

%do i=1 %to &nobs;
  %let rc =
    %sysfunc(fetchobs(&dsid, &i));

  %put %trim(&name), &age years;
%end;

%let rc = %sysfunc(close(&dsid));
```

Listing 6: SAS Macro

Note, that in both cases we just load data into variables (lua variables and macro variables respectively) and do nothing with them, since we only want to imitate a data reading situation without any further processing.

The performance profiling by ESM will look as follows:

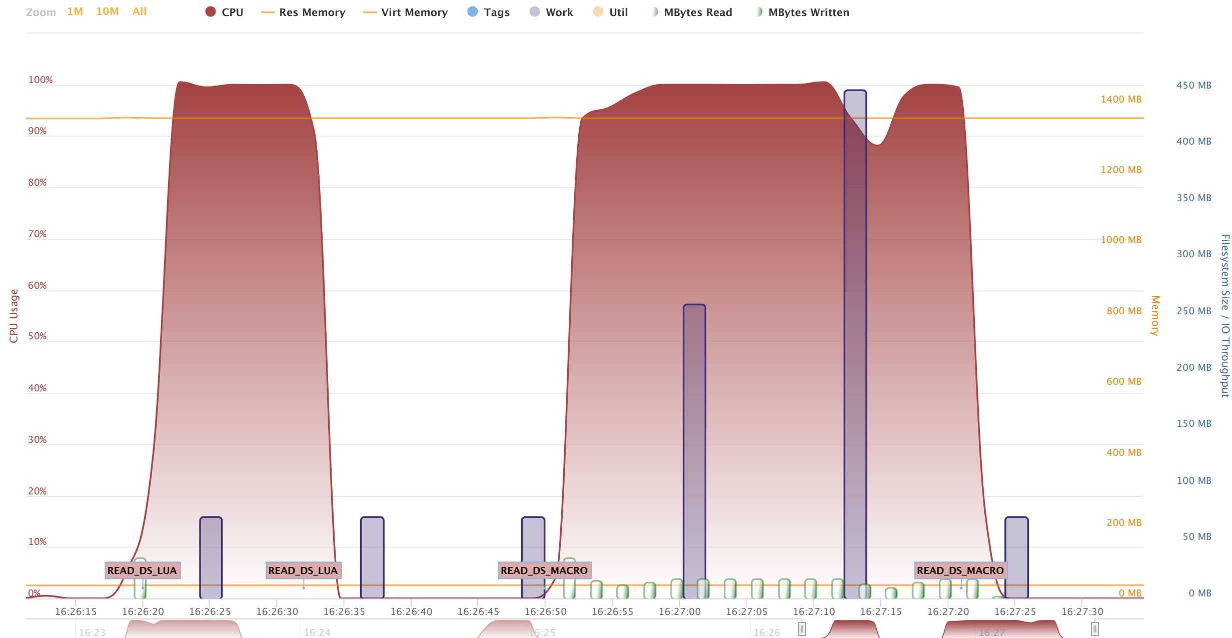

Figure 3: Reading a data set – PROC LUA on the left, SAS macro on the right.

The situation changed a little bit. But still, the Lua version looks as a strong competitor. All our examples showed that the key disadvantages of using SAS macro in terms of performance are intense disk write operations and much longer execution time.

To sum it up, PROC LUA has better performance, more natural syntax, and useful data structures.

TRANSPILING

A transpiler is a program that takes the source code of a program written in one programming language as its input and produces the equivalent source code in another programming language. In other words, it is a source-to-source compiler (like PROC DSTODS2 in SAS).

In order to produce the equivalent Lua version of SAS macro we are going to classify macros into 3 categories:

- Outside step macro – generates SAS code outside of a step boundary.
- Inside step macro – generates SAS code inside a step.
- Pure SAS macro – does not generate any SAS code, but can *return* values. Usually, that type of macro is called within SAS macro expressions.

Let's consider them in more details.

OUTSIDE STEP MACRO

That type of macro can be used only in open code outside of PROCs/DATA step boundaries. The reason for this is that these macros usually use PROCs/DATA steps internally. Let's consider this on example:

```
/* print_dataset.sas */
```

```
%macro print_dataset(dat, vars=);
  proc print data=&dat;
    %if %length(vars) %then %do;
      var &vars;
    %end;
  run;
%mend;

/* main.sas */
%print_dataset(sashelp.class, name age)
```

This macro uses PROC PRINT. Consequently, it must always be called outside of PROC and DATA steps. The following call will result in an error.

```
data _null_;
  set sashelp.class;
  %print_dataset(sashelp.cars)    <= ERROR here
  bmi = weight / height ** 2;
run;
```

It is the most convenient macro's type for transpiling into Lua. Resulting Lua equivalent would be:

```
function print_dataset(dat, vars)
  sas.submit_("proc print data=@dat@;")
  if string.len(vars) then
    sas.submit_("var @vars@;")
  end
  sas.submit("run;")
end

print_dataset("sashelp.class", "name age")
```

The key challenges here are to identify the *islands* of SAS code in between SAS Macro Statements and step boundaries (like we had above with "run;"). These islands of SAS code we are going to call **free text**.

INSIDE STEP MACRO

Inside step macro generates SAS code as well, but, this code belongs to a particular step or statement. In other words, such macro resolves into SAS code that becomes a part of a statement, PROC or DATA step. A classic example is a macro that generates attributes for variables:

```
/* attrib.sas */
%macro attrib(dat);
  %let dat = %upcase(&dat);
  %if &dat = DM %then %do;
    attrib STUDYID length=$10 label='Study Identifier'
           USUBJID length=$20 label='Unique Subject Identifier';
  %end;
  %else %if &dat = AE %then %do;
```

```
    attrib AETERM  length=$200 label='Reported Term for the Adverse Event'
           AEDECOD length=$200 label='Dictionary-Derived Term'
           AESTDTC length=$19  label='Start Date/Time of Adverse Event';
  %end;
  %else %do;
    %put ERROR: Unknown dataset!;
  %end;
%mend;

/* main.sas */
data ae;
  %attrib(ae);
  aeterm = 'Lorem ipsum dolor sit amet';
  aedecod = 'consectetur adipisicing elit.';
  aestdtc = put(today(), is8601dt.);
run;
```

The way how we transpile this kind of macro is the following:

1. transpile the macro to return generated SAS code as a string
2. save the result string to a macro variable
3. substitute that macro variable to the place where macro is called.

```
proc lua restart;
submit;
  -- (1)
  function attrib(dat)
    local r = ""
    dat = dat:upper()
    if dat == 'DM' then
      r = r .. [[
        attrib STUDYID length=$10 label='Study Identifier'
               USUBJID length=$20 label='Unique Subject Identifier';
      ]]
    elseif dat == 'AE' then
      r = r .. [[
        attrib AETERM  length=$200 label='Reported Term for the Adverse Event'
               AEDECOD length=$200 label='Dictionary-Derived Term'
               AESTDTC length=$19  label='Start Date/Time of Adverse Event';
      ]]
    else
      print('ERROR: Unknown dataset!')
    end
    return r
  end

  -- (2)
  sas.symput('attrib_1', attrib('ae'), 'G')
endsubmit;
run;

data ae;
  /* (3) */
  &attrib_1
```

```
  aeterm = 'Lorem ipsum dolor sit amet';
  aedecod = 'consectetur adipisicing elit.';
  aestdtc = put(today(), is8601dt.);
run;
```

PURE SAS MACRO

Pure SAS macro is a macro that returns text that is later used within SAS macro expressions. Let's consider a macro that checks whether a dataset or a view exists and return a 1 or 0 accordingly:

```
/* exist_dataset.sas */
%macro exist_dataset(dat);

  %if %sysfunc(exist(&dat)) or %sysfunc(exist(&dat, VIEW)) %then 1;
  %else 0;

%mend;

/* main.sas */
/* 1 */
%put {%exist_dataset(sashelp.class)};

/* 2 */
%if %exist_dataset(sashelp.class) %then %do;
  %put SASHELP.CLASS exists!;
%end;
%else %do;
  %put SASHELP.CLASS does not exist!;
%end;
```

Resulting Lua equivalent would be:

```
function exist_dataset(libds)
  if sas.exist(libds) or sas.exist(libds, 'VIEW') then
    return 1
  else
    return 0
  end
end

-- 1
print('{'..exist_dataset('sashelp.class')..'}')

-- 2
if exist_dataset('sashelp.class') then
  print('SASHELP.CLASS exists!')
else
  print('SASHELP.CLASS does not exist!');
end
```

From the examples of different macro types considered, we can see that two main points.

- Free text in outside step macro is passed into “`sas.submit`” for execution.
- Free text in inside step macro and pure macros is returned as a value and is later used from the calling environment.

The complexity in transpiling inside step macro and pure macros is that PROC LUA cannot be called at the calling point. Therefore, the calling environment must be transpiled as well, as we have seen in inside step macro example. We are not going to consider workarounds with %SYSFUNC with DOSUBL Function because of the performance implications, although this can be a solution in some situations.

FINAL LOOK OF THE TOOL

```
$ ./sasmacro2lua src/ -o lua/
[1/3] Parsing input...
        4 files
        3 macros
[2/3] Building dependecy graph...
        1 pure sas macros
        1 inside step
        1 outside step
[3/3] Transpiling...
Transpiled successfully in 1.12s.
```

ARCHITECTURE

This tool uses SAS Parser from the SASLint project[6]. The parser takes SAS program on input and produces a parse tree – a machine-readable data structure that represents a syntactic structure of a program:

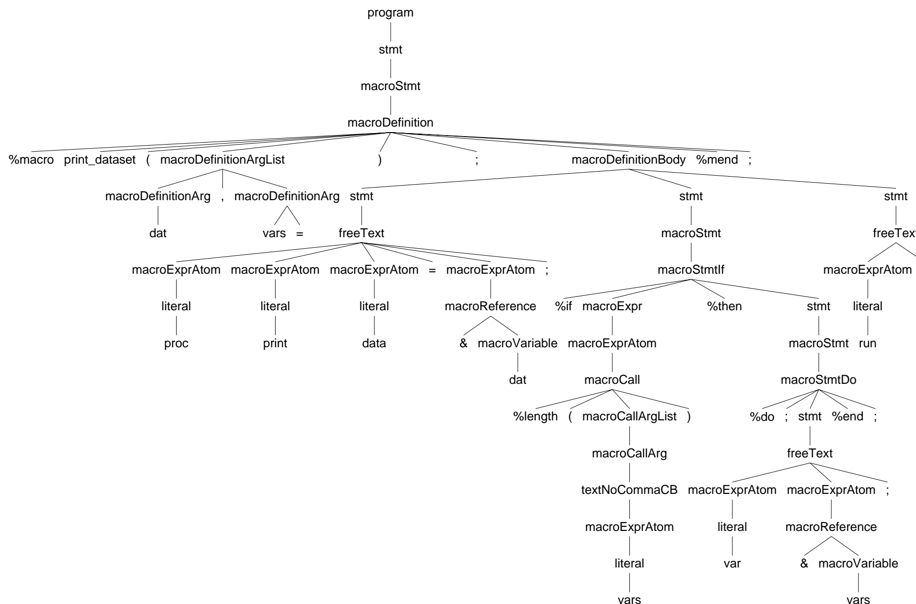

Figure 4: Parse Tree of the print_dataset.sas program from outside step macro section

[6] Khorlo (2018)

By walking this tree, we collect information out of it and build a *dependency graph* and identify the type of each macro:

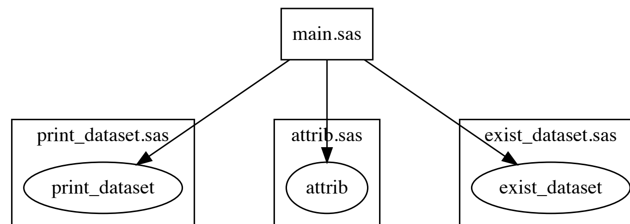

Figure 5: Dependency Graph of the input

After identifying types of the macros, we walk the tree again using the visitor pattern and output Lua statements instead of SAS Macro statements. Below is the visitor for %IF-%THEN/%ELSE Statement:

```
@Override
public String
   visitMacroStmtIfWithElse(SASMacroParser.MacroStmtIfWithElseContext ctx) {

    String evalMacroExpr = visit(ctx.macroExpr());
    String evalStmtThen = visit(ctx.stmt(0));
    String evalStmtElse = visit(ctx.stmt(1));
    String r = "if " + evalMacroExpr + " then\n" +
               "  " + evalStmtThen + "\n" +
               "else\n" +
               "  " + evalStmtElse + "\n";
    return r;
}
```

The transpiler can be tried at https://saslint.com/sasmacro2lua.

ACKNOWLEDGMENTS

I want to thank Nikola Marković who inspired myself and showed that incredible performance difference between Lua and SAS macro during the debut SAS User Group Germany Meetup in Berlin[7].

The ESM software was kindly provided by Boemska for the performance profiling purposes in the scope of this paper – this made the comparison of PROC LUA and SAS macro an easy and enjoyable experience.

CONTACT

Igor Khorlo – igor.khorlo@gmail.com

[7] SUGG (2018)

REFERENCES

Hale, Rowland (2018a): PROC LUA and why you should know it. 2018, URL: http://www.pharmasug.org/china/2018/workshop.html [Accessed: 23.3.2019]

Hale, Rowland (2018b): PROC LUA and why you should know it (workshop materials). 2018, URL: https://github.com/rowland2425/Lua_PharmaSUG_China_2018 [Accessed: 23.3.2019]

Khorlo, Igor (2018): SASLint: A SAS® Program Checker. In: *SAS® Global Forum 2018 Proceedings*, 2018, URL: https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2543-2018.pdf [Accessed: 23.3.2019]

SAS, Institute Inc. (2018): Base SAS® 9.4 Procedures Guide, Seventh Edition: LUA Procedure. 2018, URL: https://documentation.sas.com/?docsetId=proc&docsetTarget=p0lqta2cbq9b44n12h28nil7a093.htm&docsetVersion=9.4&locale=en [Accessed: 23.3.2019]

SUGG (2018): Debut SAS User Group Germany Meetup in Berlin. 2018, URL: https://www.sasusergroups.org/meetup/sugg/2018/09/14/sugg-debut-meetup-berlin.html [Accessed: 23.3.2019]

Tomas, Paul (2015): Driving SAS® with Lua. In: *SAS® Global Forum 2015 Proceedings*, 2015, URL: https://support.sas.com/resources/papers/proceedings15/SAS1561-2015.pdf [Accessed: 23.3.2019]