Five Reasons to Swipe Right on PROC FCMP

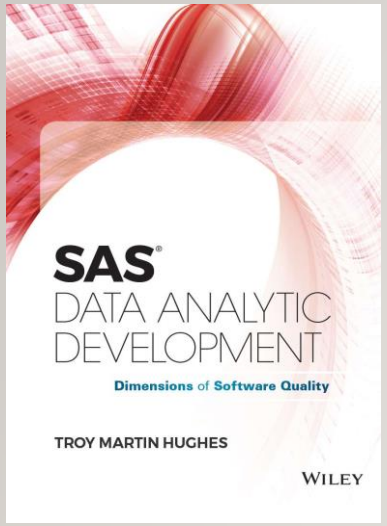
The SAS® Function Compiler for Building User-Defined Functions

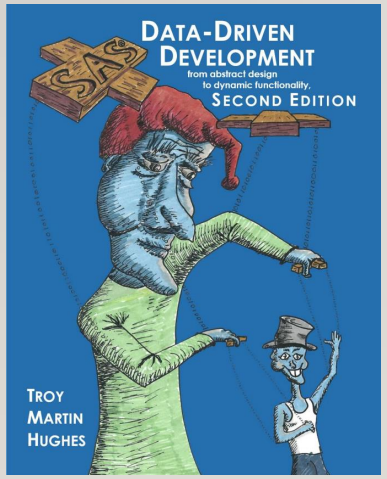
Boston Area SAS User's Group (BASUG)

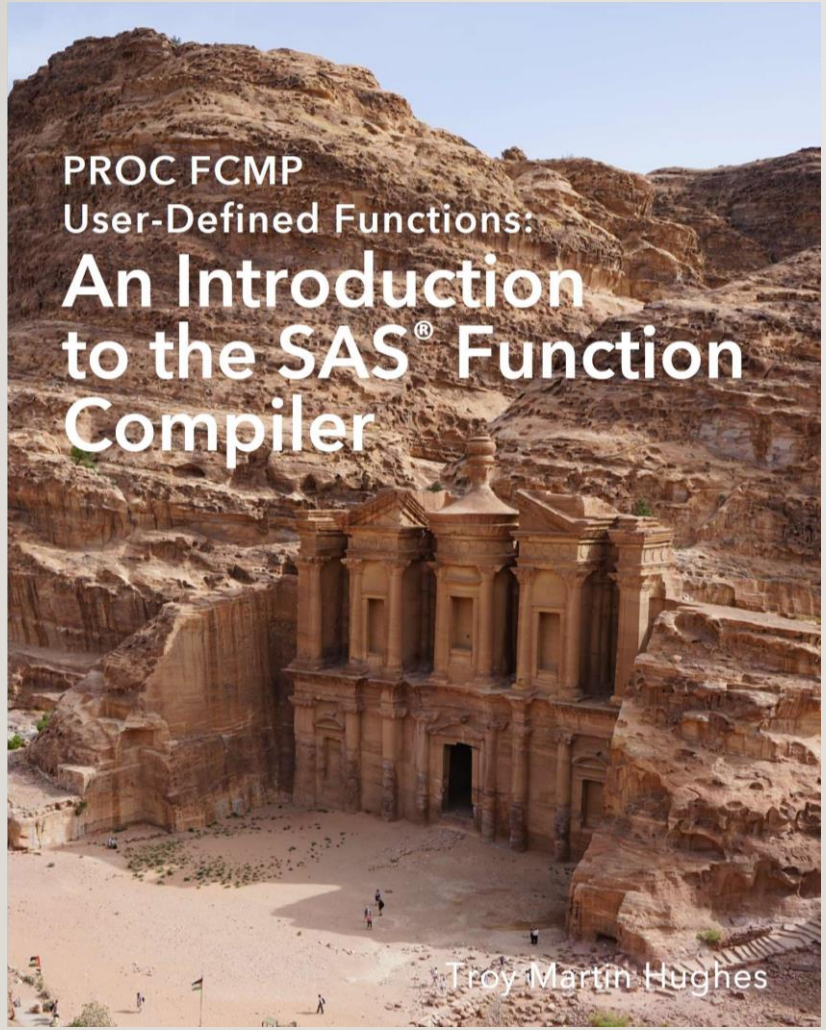
Troy Martin Hughes

June 7, 2023

BIOGRAPHY







Troy has been a SAS practitioner for more than 20 years, has managed SAS projects in support of federal, state, and local government initiatives, and is a SAS Certified Base, Advanced, V8, and Clinical Trials Programmer. Since 2013, he has presented more than 150 talks, trainings, and posters at SGF, SAS Analytics Experience, WUSS, SCSUG, MWSUG, SESUG, PharmaSUG, BASAS, and BASUG. He has an MBA in Information Systems Management and certifications including: PMP, PMI-RMP, PMI-PBA, PMI-ACP, SSCP, CSSLP, CISSP, CRISC, CISA, CGEIT, CISM, CSM, CSD, A-CSD, CSPO, CSP-SM, CSP-PO, CSP, SAFe Government Practitioner, Network+, Security+, CySA+, CASP+, Cloud+, and ITIL Foundation. Troy is a US Navy veteran with two Afghanistan deployments.

OUTLINE

- Five Solutions Using FCMP User-Defined Functions:
  1. incorporate and hide (encapsulate) complex operations (like hash objects)
  2. manipulate arrays through reusable methods
  3. execute complex operations during format/informat application
  4. execute DATA steps and PROCs from inside DATA steps (e.g., getters/setters)
  5. replace unnecessary reliance on user-defined macros and metaprogramming
- Five High-Level Benefits of User-Defined Functions:
  1. software modularity
  2. software maintainability
  3. software reusability
  4. software security
  5. software configurability

OBJECTIVES

1. Introduce functions and function nomenclature from a software-agnostic perspective, and demonstrate how functions increase software *modularity*, *maintainability*, *reusability*, *security*, and *configurability*.
2. Introduce the FCMP procedure, and demonstrate FCMP use cases and basic FCMP syntax.
3. Ultimately, improve 1) the quality of the software you write and 2) the quality of the development environment in which you write software.

Productivity  and Popularity 

Nomenclature

Software Quality Nomenclature

Some dimensions of software quality:

- **modularity** – decomposition of software into discrete components, such that modification to one component requires minimal or no change to others
- **maintainability** – the ability for a system or software to be modified (while in production) to correct defects, alter functionality, or improve performance
- **reusability** – the ease with which a software component can be reused, either in the current or future software products or projects
- **security** – code defined and maintained within functions is independently tested, stored, and accessed, which mitigates accidental corruption
- **configurability** – the ability for a system or software to produce variable output or functionality when provided variable input

BUILT-IN VS. USER-DEFINED

Built-in – a software component that ships (or downloads) with a software application, such as SAS built-in functions, subroutines, or procedures:

- PROC SORT – built-in procedure that orders a data set
- %LENGTH – built-in macro function that evaluates length of text
- CALL SORT – built-in subroutine that orders variables or values
- UPCASE – built-in function that capitalizes a character variable

User-Defined – a software component that is built, tested and documented (hopefully), and deployed by end users (i.e., that's us!)

Calls and Callable Modules

Call

- “1. transfer of control from one software module to another, usually with the implication that control will be returned to the calling module”
- “3. to transfer control from one software module to another as in (1) and, often, to pass parameters to the other module”

- ISO/IEC/IEEE 24765:2017-09, second edition

Callable Module – a module capable of being called (by invoking its name)

Calling Module – the program or module that calls (and temporarily transfers program control to) a callable module

Called Module – the module that is called by the calling module

FUNCTIONS

Function

- “1. defined objective or characteristic action of a system or component”
- “2. software module that performs a specific action, is invoked by the appearance of its name in an expression, receives input values, and returns a single value”

  - ISO/IEC/IEEE 24765:2017-09, second edition

- some SAS built-in functions (e.g., DATE, TIME) do not declare parameters, and instead rely on SAS or system environment variables for all input
- SAS functions—both built-in and user-defined—will nearly always require input

Function Components

Specification (aka, the specs) – defines the function's functionality, its input (parameter declaration), its output (return value, return code), and clarifying context and caveats for its usage:

- during software development, the "specs" refers to the set of functional and performance requirements that the software product must meet
- during software operation (O&M), the "specs" refers to the function description, including external documentation or comments inside the program

Implementation (aka, the function definition) – code that executes when the function is called

Invocation (aka, the function call) – statement that invokes a function (by calling its name), and temporarily transfers program control to the function

FUNCTION COMPONENTS: LOWCASE BUILT-IN FUNCTION

specification, implementation, and invocation

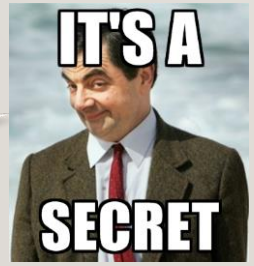
invocation

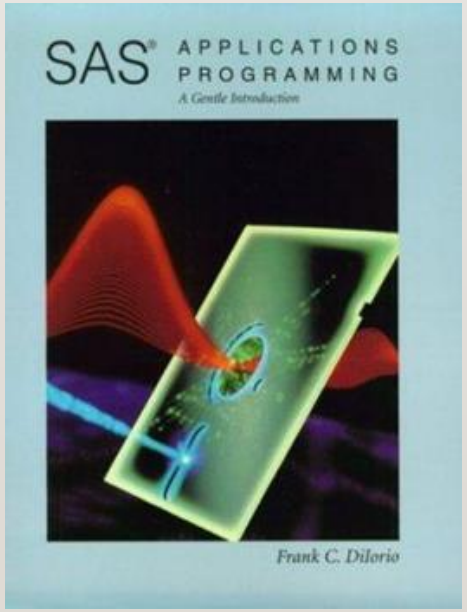
specification

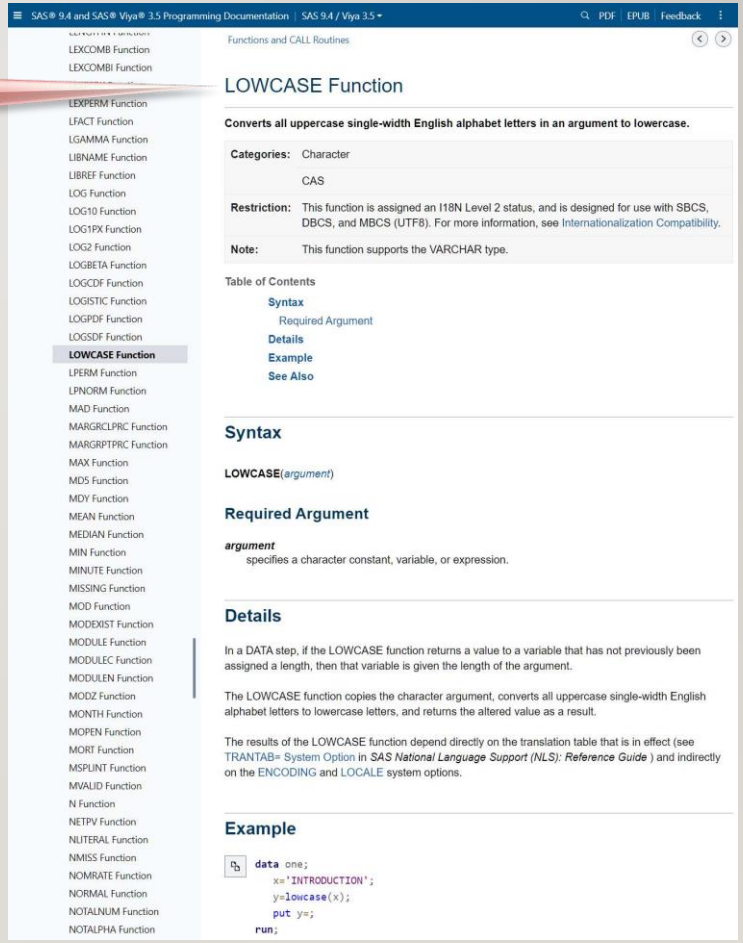
```
data lower;
   length phrase $100;
   phrase = 'SAS Applications Programming:
      A Gentle Introduction';
   phrase=lowcase(phrase);
   put phrase;
run;

sas applications programming:
 a gentle introduction
```

implementation







Let's Travel Back in Time

Reverse Engineering a Built-in Function

Reinventing the wheel *accidentally* is fruitless; however, **intentional reverse engineering of a built-in function can be beneficial**, especially when learning how to build user-defined functions in a programming language:

- testing a user-defined function is simpler because the output of your user-defined functions can be compared to the output (i.e., return value or return code) of its archetypal built-in function—to validate both *use cases* and *misuse cases*
- built-in functions typically evince complex, comprehensive *exception handling* that imbues robustness, enabling a function to overcome (or at least detect) unwanted variability; this exception handling can be emulated in user-defined functions
- built-in functions often produce notes or warnings in the log, both of which can be produced by user-defined SAS functions
- built-in functions are documented extensively; user-defined functions should be as well

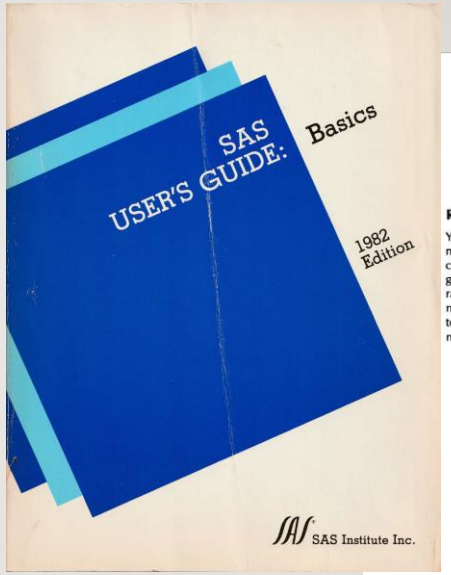
A TIME BEFORE LOWCASE

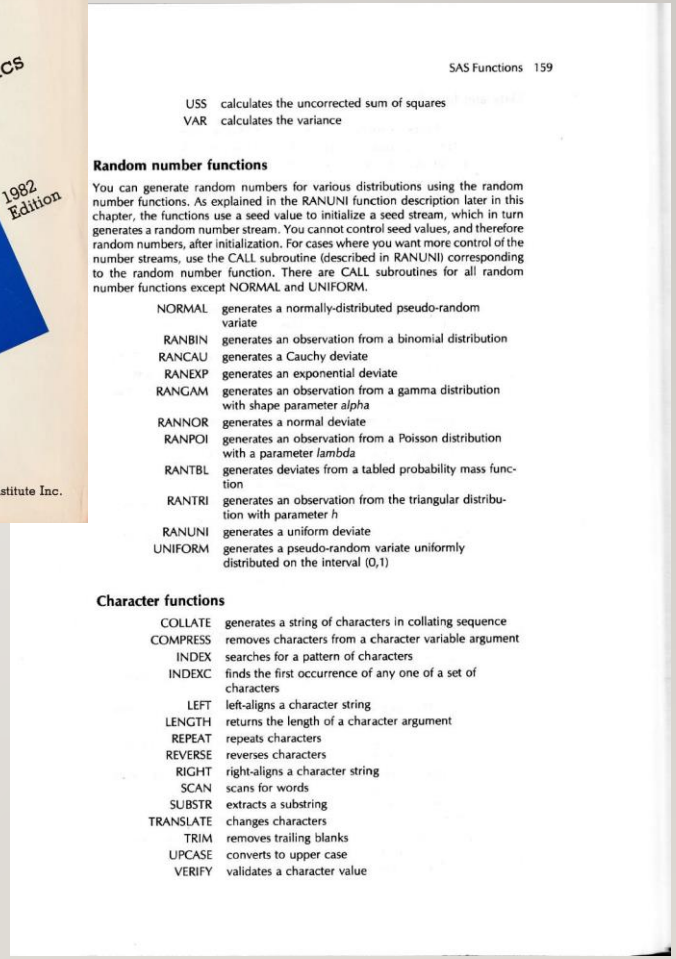
On page 353, Frank notes:

> **UPCASE.** UPCASE converts all lowercase characters in its argument to uppercase. Its syntax follows:
>
> **`UPCASE(string)`**
>
> The *string* is a character string. Nonalphabetic characters are unaffected by this function. There is no analogous function to convert to lowercase.

- *SAS Applications Programming: A Gentle Introduction*

1982 SAS User's Guide contains only 15 built-in character functions!!!





SAS Functions 159

USS	calculates the uncorrected sum of squares
VAR	calculates the variance

Random number functions

You can generate random numbers for various distributions using the random number functions. As explained in the RANUNI function description later in this chapter, the functions use a seed value to initialize a seed stream, which in turn generates a random number stream. You cannot control seed values, and therefore random numbers, after initialization. For cases where you want more control of the number streams, use the CALL subroutine (described in RANUNI) corresponding to the random number function. There are CALL subroutines for all random number functions except NORMAL and UNIFORM.

NORMAL	generates a normally-distributed pseudo-random variate
RANBIN	generates an observation from a binomial distribution
RANCAU	generates a Cauchy deviate
RANEXP	generates an exponential deviate
RANGAM	generates an observation from a gamma distribution with shape parameter *alpha*
RANNOR	generates a normal deviate
RANPOI	generates an observation from a Poisson distribution with a parameter *lambda*
RANTBL	generates deviates from a tabled probability mass function
RANTRI	generates an observation from the triangular distribution with parameter *h*
RANUNI	generates a uniform deviate
UNIFORM	generates a pseudo-random variate uniformly distributed on the interval (0,1)

Character functions

COLLATE	generates a string of characters in collating sequence
COMPRESS	removes characters from a character variable argument
INDEX	searches for a pattern of characters
INDEXC	finds the first occurrence of any one of a set of characters
LEFT	left-aligns a character string
LENGTH	returns the length of a character argument
REPEAT	repeats characters
REVERSE	reverses characters
RIGHT	right-aligns a character string
SCAN	scans for words
SUBSTR	extracts a substring
TRANSLATE	changes characters
TRIM	removes trailing blanks
UPCASE	converts to upper case
VERIFY	validates a character value

A TIME BEFORE LOWCASE

Non-callable LOWCASE functionality:

```
data lower;
   length phrase $100;
   phrase = 'SAS Applications Programming: A
Gentle Introduction';
   do i = 1 to length(phrase);
      if 65 <= rank(char(phrase,i)) <= 90
         then substr(phrase,i,1)
         = byte(rank(char(phrase,i)) + 32);
      end;
   put phrase;
run;


sas applications programming: a gentle
introduction
```

Software Quality Characteristics Checklist

- ✖ **modularity**
- ✖ **maintainability**
- ✖ **readability**
- ✖ **reusability**
- ✖ **security**
- ✖ **portability**
- ✖ **configurability**

A TIME BEFORE LOWCASE

Callable LOWCASE functionality:

specification

implementation

invocation

```
proc fcmp outlib=work.funcs.char;
   function tiny(str $) $100;
      do i = 1 to length(str);
         if 65 <= rank(char(str,i)) <= 90
            then substr(str,i,1)
            = byte(rank(char(str,i)) + 32);
         end;
      return(str);
      endfunc;
quit;

options cmplib=work.funcs;

data lower;
   length phrase $100;
   phrase = tiny('The Little SAS Book');
   put phrase;
run;
```

TINY function:

- created 6-7-2023, tmh
- lowers case of a character variable
- argument cannot exceed 100 characters
- no warning for truncation
- tested on Windows
- not portable to mainframe

A TIME BEFORE LOWCASE

Callable LOWCASE functionality:

```
proc fcmp outlib=work.funcs.char;
   function tiny(str $) $100;
      do i = 1 to length(str);
         if 65 <= rank(char(str,i)) <= 90
            then substr(str,i,1)
            = byte(rank(char(str,i)) + 32);
      end;
      return(str);
      endfunc;
quit;

options cmplib=work.funcs;

data lower;
   length phrase $100;
   phrase = tiny('The Little SAS Book');
   put phrase;
run;
```

FCMP wrapper

function declaration

function name

STR character parameter

terminate PROC FCMP

output location

declare char return value

return value

end function declaration

reference output location

1. Hide Your Hash

Hide Hash and Other Complex Operations

Requirements:

- perform a horizontal sort of character values within a single character variable
- simplify (and modularize) code by encapsulating hash operations

Featured in:

- *Sorting a Bajillion Variables: When SORTC and SORTN Subroutines Have Stopped Satisfying, User-Defined PROC FCMP Subroutines Can Leverage the Hash Object to Reorder Limitless Arrays*

"Horizontal" Sort – Reorder Values (in a Variable)

We the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America.

"Horizontal" Sort – Reorder Values (in a Variable)


```
a a a a a a a a a a a a
a a a a a a a a a a a a
a a a a a a a a a a a a
a a a a a a a a a a a a
a a a a ability absence
absent absolutely accept
acceptance according
according according
according accordingly
account act act act acts
acts actual actual
actually
```

HASH OBJECT "HORIZONTAL" SORT (OF CHARACTER VAR)

```
proc fcmp outlib=work.funcs.sort;
   subroutine reorder_words(string $);
      outargs string;
      length key $20 num cnt 8;
      declare hash h(ordered:
         'ascending');
      declare hiter iter('h');
      rc=h.defineKey('key');
      rc=h.defineData('key','cnt');
      rc=h.defineDone();
      do num=1 to countw(string);
         key=scan(string,num,'','S');
         * increment/add a key;
         if h.find()=0 then cnt+1;
         else cnt=1;
         rc=h.replace();
         end;
         * rebuild ordered char var;
         num=1;
         string='';
         do while(iter.next()=0);
            do tot=1 to cnt;
               string=catx(' ',
                  string,key);
               num+1;
               end;
            end;
         endsub;
quit;
```

HASH OBJECT "HORIZONTAL" SORT – INVOCATION

Calling the REORDER_WORDS subroutine:

```
data _null_;
   length some_words $100;
   some_words='curry petra cheap friends inside of abdullah
      aqib bought for and';
   call reorder_words(some_words);
   put some_words=;
run;
```

Results printed to log:

```
some_words=abdullah and aqib bought cheap curry for friends inside
of petra
```

2. Manipulate Arrays

Manipulate Arrays and Hide Complex Operations

Requirements:

- perform a horizontal sort of character variables
- design a callable module that overcomes an 800-variable limitation of CALL SORTC (and all other built-in functions that leverage the OF operator inside PROC FCMP)
- simplify (and modularize) code by encapsulating hash and array operations

Featured in:

- *Sorting a Bajillion Variables: When SORTC and SORTN Subroutines Have Stopped Satisfying, User-Defined PROC FCMP Subroutines Can Leverage the Hash Object to Reorder Limitless Arrays*

"Horizontal" Sort – Reorder Variables


```
We
the
People
of
the
United
States,
in
Order
to
form
a
more
perfect
Union,
establish
Justice,
```

"Horizontal" Sort – Reorder Variables

a
a
a
a
a
a
a
a
a
a
a
a
a
a
a
a

"Horizontal" Sort – Reorder Variables

Typically performed using:

1. SORT function or subroutine
2. SORTC function or subroutine (for character variables)
3. SORTN function or subroutine (for numeric variables)

```
proc transpose data=const
      out=const_wide
      prefix=word;
   var word;
run;


data const_wide_sorted;
   set const_wide
      (drop=_name_);
   array words[*] $ word:;
   call sortc(of words[*]);
   put word1-word7;
run;
```

HASH OBJECT "HORIZONTAL" SORT (OF ARRAY)

```
proc fcmp outlib=work.funcs.sort;
   subroutine reorder(arr[*] $);
      outargs arr;
      length key $20 num cnt 8;
      declare hash h(ordered:
         'ascending');
      declare hiter iter('h');
      rc=h.defineKey('key');
      rc=h.defineData('key','cnt');
      rc=h.defineDone();
      do num=1 to dim(arr);
         key=arr[num];
         * increment/add a key;
         if h.find()=0 then cnt+1;
         else cnt=1;
         rc=h.replace();
         end;
      * build ordered array;
      num=1;
      do while(iter.next()=0);
         do tot=1 to cnt;
            arr[num]=key;
            num+1;
            end;
         end;
      endsub;
quit;
```

Hash Object "Horizontal" Sort – Invocation

Calling the REORDER subroutine:

```
data const_wide_sorted;
   set const_wide (drop=_name_);
   array words[*] $ word:;
   call reorder(of words);
   put word1-word80;
run;
```

Results printed to log:

```
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a ability absence absent absolutely accept
acceptance according according according according accordingly account act
act act acts acts actual actual actually adding adhering adjourn adjourn
adjourn adjournment adjournment adjournment
NOTE: There were 1 observations read from the data set WORK.CONST_WIDE.
NOTE: The data set WORK.CONST_WIDE_SORTED has 1 observations and 4422
variables.
```

3. Use Functions as Formats

Apply a Format/Informat That Calls a Function

Requirements:

- perform data validation during data ingestion (i.e., INPUT statement)
- facilitate complex business rules and logic inside informat application

Featured in:

- *Make You Holla' Tikka Masala: Creating User-Defined Informats Using the PROC FORMAT OTHER Option To Call User-Defined FCMP Functions That Facilitate Data Ingestion Data Quality*

Master Data (Lookup Table)

Establish a core set of potential tikka masala ingredients:

```
data ingredients;
   infile datalines dsd;
   length ingredient $50;
   input ingredient $;
   datalines;
cumin
chili powder
coriander
turmeric
garlic
salt
chicken
;
```

In master data management (MDM), these integral, unduplicated data can be termed "master data" and in aggregate can be referred to as a "master table" (aka, the *golden record* or *golden table*)

Traditional Method: User-Defined Informat

CNTLIN option in PROC FORMAT can be used to create data-driven formats and informats:

```
set ingredients(rename=(ingredient=start)) end=eof;
   length label $32 fmtname $32 type $2 hlo $2;
   label=start;
   fmtname='inf_tikka';
   type='j';       * J denotes a character informat;
   hlo='';
   output;
   if eof then do;
      label='';
      start='';
      hlo='o';     * O denotes other (i.e., invalid);
      output;
      end;
run;

proc format cntlin=inf_tikka_data;
run;
```

User-Defined Function (with Hash Object)

By placing the hash table and its functionality inside a user-defined function (defined using PROC FCMP), hash functionality is encapsulated, and becomes callable and reusable:

```
proc fcmp outlib=work.funcs.recipes;
   function make_you_holla_tikka_masala(ingredient $) $;
      declare hash h(dataset: 'ingredients');
      rc=h.defineKey('ingredient');
      rc=h.defineDone();
      if h.check()=0 then return(ingredient);
      else return('');
      endfunc;
quit;
```

	item	quantity	measurement
1	turmeric	1	speck
2	garlic	5	cloves
3	chicken	8	breasts
4	cumin	1	bounty
5	coriander	20	seeds
6	salt	2	pinches
7		2	pumps
8		3	packages
9		8	handfuls

The CMPLIB system option must be specified prior to calling a user-defined function:

```
options cmplib=work.funcs;

data troys_tasty_tikka_valid_hash;
   set troys_tasty_tikka;
   item=make_you_holla_tikka_masala(item);
run;
```

User-Defined Informat That Calls a Function

With a user-defined function created, PROC FORMAT can call that function, which will perform function operations whenever the informat is applied to a variable:

```
proc format;
   invalue $ inf_tikka_hash other=[make_you_holla_tikka_masala()];
run;
```

When the informat is applied to the ingredients, the identical data set is created:

```
data troys_tasty_tikka_valid;
   infile f dsd delimiter=',';
   length item $50 quantity 8 measurement $ 20;
   input item : $inf_tikka_hash. quantity measurement $;
run;
```

	item	quantity	measurement
1	turmeric	1	speck
2	garlic	5	cloves
3	chicken	8	breasts
4	cumin	1	bounty
5	coriander	20	seeds
6	salt	2	pinches
7		2	pumps
8		3	packages
9		8	handfuls

One downside, however, is that when OTHER is utilized to call the function, it can no longer initialize the _ERROR_ automatic variable; a workaround is demonstrated in the white paper.

4. Deep PROC and Deep DATA

RUN A DATA STEP OR PROC INSIDE A DATA STEP

Requirements:

- calculate average diameter of planets in solar system
- perform calculations dynamically (only if needed during runtime)
- perform calculations inside a step that *traditionally* would require a preceding PROC or DATA step

Featured in:

- *Undo SAS® Fetters with Getters and Setters: Supplanting Macro Variables with More Flexible, Robust PROC FCMP User-Defined Functions That Perform In-Memory Lookup and Initialization Operations*

Create Master Data (Lookup Table)

Create the Planets data set (with diameters in kilometers):

```
data planets;
   infile datalines dsd delimiter=',';
   length name $10 diameter 8;
   input name $ diameter;
   format diameter comma8.0;
   datalines;
mercury,4879
venus,12104
earth,12756
mars,6792
jupitor,142984
saturn,120536
uranus,51118
neptune,49528
pluto,2376
;
```

FUNCTION CALLS MACRO THAT CALLS PROC SQL

The COMPUTE_MEAN_MACRO macro executes PROC SQL:

```
%macro compute_mean_macro();
%let dsn=%sysfunc(dequote(&dsn));
%let var=%sysfunc(dequote(&var));
proc sql noprint;
   select avg(&var) into: avg
      from &dsn;
quit;
%mend;
```

The COMPUTE_MEAN function calls the COMPUTE_MEAN_MACRO macro:

```
proc fcmp outlib=work.funcs.stats;
   function compute_mean(dsn $,var $);
      rc=run_macro('compute_mean_macro', dsn, var, avg);
      return(avg);
      endfunc;
quit;
```

CALL FUNCTION THAT INVOKES RUN_MACRO

The DATA step calls the COMPUTE_MEAN function:

```
data _null_;
   set planets;
   length statement $40 diff 8;
   diff=diameter - compute_mean('planets','diameter');
   if diff>0 then statement=strip(name) || ' is ' ||
strip(put(diff,comma8.)) ||
      'km greater than average';
   else statement=strip(name) || ' is ' ||
strip(put(abs(diff),comma8.)) ||
      'km less than average';
   put statement;
run;
```

Log:

```
mercury is 39,907km less than average
venus is 32,682km less than average
earth is 32,030km less than average
```

FUNCTION CALLS MACRO THAT CALLS DATA STEP

The COMPUTE_MEAN_MACRO macro executes the DATA step:

```
%macro compute_mean_macro();
%let dsn=%sysfunc(dequote(&dsn));
%let var=%sysfunc(dequote(&var));
data _null_;
   set &dsn end=eof;
   retain tot 0;
   tot=sum(tot,&var);
   if eof then call symputx('avg',tot/_n_,'g');
run;
%mend;
```

The COMPUTE_MEAN function calls the COMPUTE_MEAN_MACRO macro :

```
proc fcmp outlib=work.funcs.stats;
   function compute_mean(dsn $,var $);
      rc=run_macro('compute_mean_macro', dsn, var, avg);
      return(avg);
      endfunc;
quit;
```

5. Replacing Metaprogramming

Replacing Unnecessary Macro-Based Metaprogramming

Requirements:

- dynamically build a hash object that performs lookup operations
- maintain data exclusively within a built-in data structure (i.e., SAS data set) to ensure special characters are not encoded within macro variables
- design a solution flexible enough to accommodate various hash objects

Featured in:

- *Picking Scabs and Digging Scarabs: Refactoring User-Defined Decision Table Interpretation Using the SAS® Hash Object To Maximize Efficiency and Minimize Metaprogramming*

  (authored with Louise Hadden aka Lil Weezie aka Mainframe Mama aka the Girl with the SAS tattoo!)

METAPROGRAMMING WITH SAS MACRO LANGUAGE

```
%let loc=C:\sas\;
data rum_to_tools;
   infile "&loc.rum_tool_table.csv" truncover firstobs=1 end=eof;
   length line $10000 decision_point_var outcome_var inp_val out_val $32
      macro_code $10000;
   format line $10000.;
   retain decision_point_var outcome_var macro_code '';
   input line & $;
   if _n_=1 then do;
      decision_point_var=strip(scan(line,1,','));
      outcome_var=strip(scan(line,2,','));
      end;
   else do;
      inp_val=strip(scan(line,1,','));
      out_val=strip(scan(line,2,','));
      if _n_>2 then macro_code=catx('',macro_code,'else');
      macro_code=strip(macro_code) || ' if ' || strip(decision_point_var)
         || '="' || strip(inp_val) || '" then '
         || strip(outcome_var) || '="' || strip(out_val) || '";';
      end;
   if eof then call symputx('macro_code',macro_code,'g');
run;
```

Master Data (Decision Table)

- Archaeologists care about soil content when selecting tools; thus, a unique combination of soil content and rum level (i.e., two decision points) will prescribe tool selection (i.e., the decision outcome).
- For example, in “potsherds” with “some” level of One Barrel rum, I’ll select a trowel.

	A	B	C
1	soil_content	rum_level	tool
2	dirt	none	N/A
3	dirt	some	shovel
4	dirt	lots	shovel
5	potsherds	none	pickaxe
6	potsherds	some	trowel
7	potsherds	lots	shovel
8	pottery	none	toothbrush
9	pottery	some	pickaxe
10	pottery	lots	trowel

A Hardcoded Function Shows Needed Abstraction

The SELECT_TOOL function has numerous hardcoded references to my decision table:

```
proc fcmp outlib=work.myfuncs.decision;
   function select_tool(soil_content $, rum_level $) $;
      length tool $100;
      declare hash h (dataset: 'rum_soil_tool_table');
      rc=h.defineKey('soil_content','rum_level');
      rc=h.defineData('tool');
      rc=h.defineDone();
      rc=h.find();
      return(tool);
      endfunc;
quit;
```

Three SQL procedures (shown in the referenced white paper) are required to instead generate these values dynamically, which are saved to macro variables.

The Dynamic Function Shows Abstraction

The updated function is now dynamically generated, providing configurability for users:

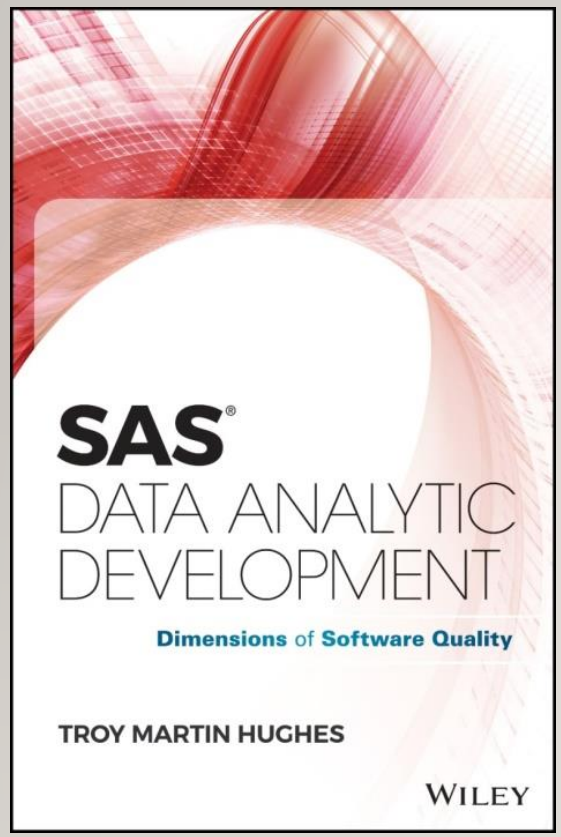
```
proc fcmp outlib=work.myfuncs.decision;
   function &function(&param_list) $;
      length &outcome_var $100;
      declare hash h (dataset: "&dsn");
      rc=h.defineKey(&key_list);
      rc=h.defineData("&outcome_var");
      rc=h.defineDone();
      rc=h.find();
      return(&outcome_var);
      endfunc;
quit;
%mend;
```

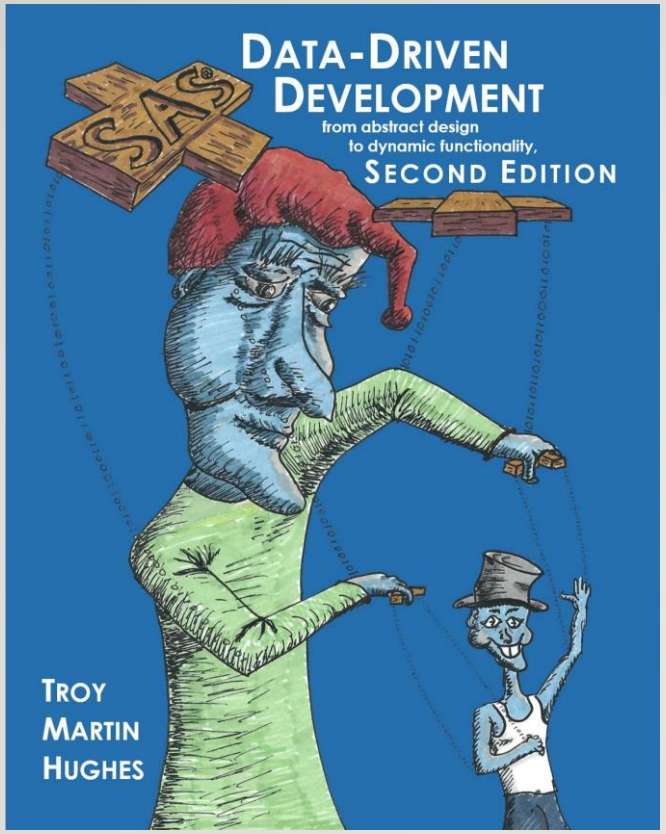
The macro is called to compile the user-defined function, after which the function can be called within a DATA step:

```
%make_decision_function(select_tool, rum_soil_tool_table);
```

Conclusion

- PROC FCMP empowers users to build user-defined functions and subroutines
- User-defined functions improve the quality of SAS software (e.g., modularity, maintainability, readability, reusability, security, portability, configurability)
- Functions also improve the quality of your development environment and experience
- Please explore the following white papers for further context and full code:
  - *Sorting a Bajillion Variables: When SORTC and SORTN Subroutines Have Stopped Satisfying, User-Defined PROC FCMP Subroutines Can Leverage the Hash Object to Reorder Limitless Arrays*
  - *Make You Holla' Tikka Masala: Creating User-Defined Informats Using the PROC FORMAT OTHER Option To Call User-Defined FCMP Functions That Facilitate Data Ingestion Data Quality*
  - *Undo SAS® Fetters with Getters and Setters: Supplanting Macro Variables with More Flexible, Robust PROC FCMP User-Defined Functions That Perform In-Memory Lookup and Initialization Operations*
  - *Picking Scabs and Digging Scarabs: Refactoring User-Defined Decision Table Interpretation Using the SAS® Hash Object To Maximize Efficiency and Minimize Metaprogramming*





troymartinhughes@gmail.com
https://www.linkedin.com/in/troy-hughes-27a998a8