Paper CC-230

# Flexible Programming With Hash Tables

Joe Matise, NORC at the University of Chicago

Code available at https://github.com/snoopy369/SESUG-2016

## ABSTRACT

When developing a general application, it often pays to use flexible techniques that will enable an application to handle a diverse set of inputs with a simple and concise syntax, while efficiently producing a consistent final output. This paper will show intermediate to advanced SAS® programmers how to make use of hash tables to produce powerful applications that are easy to use.

The example application, a data cleaning program, will enable users to specify connections between different datasets using a straightforward syntax through the use of hash tables. We will also introduce the concept of hash tables for users who are unfamiliar with hash tables and their use in SAS programming.

Before reading this paper, users should have a good understanding of the SAS data step, should have some familiarity with basic macro programming, should be familiar with data driven programming techniques, and should be comfortable combining multiple tables.

## INTRODUCTION

Designing applications for performing tasks such as data cleaning, reporting, or ETL, typically involves tradeoffs between writing programs that are flexible and writing programs that are efficient. Particularly when your program must combine data from multiple datasets together, writing an application that can process data efficiently while still maintaining the flexibility to deal with many different problems can be difficult.

When a project's data is stored in a relational format, with data in many different tables, tasks such as cleaning take on an even more complex dimension. No longer can the programmer write simple lines of code, like so:

```
if paym_amount > bill_amount then flag_error=1;
```

Instead, the tables must be combined in some fashion. While many methods exist for performing that combination – data step merge, PROC SQL join, formats, etc. – the method that will be considered here is the data step hash table. The hash table permits the developer to write a simple application that is efficient and yet capable of performing many different types of comparison.

## THE SAMPLE DATA

The project we will use as an example in this program has a few tables related to billing and payments for a healthcare firm. We will have four sample datasets: `payments`, `payments_source`, `bills`, and `bill_types`, listed below.

```
data payments;
  length patient_id $4;
  input patient_id $ paym_num paym_bill paym_source paym_amount
paym_date;
datalines;
P001 1 1 1 30.00 16FEB2015
P001 2 2 1 30.00 10MAR2015
P001 3 3 1 60.00 21MAY2015
P002 1 1 1 10.00 15JAN2015
P002 2 1 2 10.00 02FEB2015
P002 3 2 1 15.00 19APR2015
P002 4 2 2 20.00 01APR2015
P003 1 1 3 100.00 01MAR2015
P003 2 2 3 75.00 18APR2015
;;;;
run;
```

```
data payments_source;
  length paym_source_label $40;
  input paym_source_num paym_source_label $ &;
datalines;
1 Patient
2 Medicare Supplement
3 Patient Family Member
;;;;
run;

data bills;
  length patient_id $4;
  input patient_id $ bill_num bill_visit bill_type bill_amount
bill_insurance_max;
datalines;
P001 1 1 1 50.00 30.00
P001 2 2 1 50.00 30.00
P001 3 3 1 80.00 50.00
P002 1 1 2 50.00 20.00
P002 2 1 2 50.00 25.00
P002 3 2 2 80.00 35.00
P002 4 2 2 70.00 35.00
P003 1 1 1 100.00 120.00
P003 2 2 1 90.00 70.00
;;;;
run;

data bill_types;
  length bill_type_label $40;
  input bill_type_num bill_type_label & $;
datalines;
1 Regular Bill
2 Medicare Split Bill
;;;;
run;
```

## HASH TABLES

Hash tables are a kind of data step component object. Their syntax will be familiar to those comfortable with object oriented programming; they have methods and attributes, and use dot notation to refer to them.

At their simplest, hash tables are a way of storing data in memory in a quickly accessible format. They are stored in a binary tree structure, which yields very fast access to any specific data element. They interact with the data step directly, making them very easy to adapt to SAS programs.

To use a hash table, you must first declare it. Then, you must define two attributes: its *keys*, and its *data*. Finally , you end the declaration section. For example:

```
data check_payments;
  set payments;
  if _n_ = 1 then do;
    call missing(of bill_num);
    declare hash bills(dataset:'bills',multidata:'y');
    bills.defineKey('patient_id','bill_num');
    bills.defineData('bill_num');
    bills.defineDone();
  end;
run;
```

In this example, we also pre-load data into the hash using the *dataset* option, and tell SAS that the hash is allowed to have multiple rows per unique key value pair (`multidata:'y'`). We also include a *call missing* line; without it, SAS will throw an error if `bill_num` is not defined in the program anywhere other than in the hash constructor.

#### QUERYING THE HASH TABLE

The hash table then can be queried using the `find()` method. This method returns a return code, 0 if a record is found, non-zero if not. If a record is found, the data defined in `defineData()` will be placed in variables of the same name on the PDV and are immediately available to be queried.

```
data check_payments;
  set payments;
  if _n_ = 1 then do;
    call missing(of bill_num);
    declare hash bills(dataset:'bills',multidata:'y');
    bills.defineKey('patient_id','bill_num');
    bills.defineData('bill_num');
    bills.defineDone();
  end;
  rc = bills.find(key:patient_id, key:paym_bill);
  rule_1 = paym_bill = bill_num;
  if not rule_1 then
    error "PAYMENT has a missing BILL record";
run;
```

If the `find()` method is used with no arguments, SAS assumes there are variables with the same name as the keys defined in `defineKey()` containing the keys to look up; if this is not true, either the key values or variable names containing the key values need to be passed as arguments (as here).

#### WHY USE HASH TABLES?

The hash table, in addition to being very fast, is also very flexible. It can be used to perform operations such as summing values across multiple observations within a single datastep. For example:

```
data test;
  set bills;
  if 0 then set payments;
  if _n_=1 then do;
    declare hash paym(dataset:'payments',multidata:'yes');
    paym.defineKey('patient_id','paym_bill');
    paym.defineData('paym_bill','paym_amount');
    paym.defineDone();
  end;
  rc = paym.find(key:patient_id, key:bill_num);
  do while (rc=0);
    paym_sum = sum(paym_sum,paym_amount);
    rc = paym.find_Next(key:patient_id, key:bill_num);
  end;
run;
```

This allows an application developer to write modules to perform different operations knowing that they will work directly in line and will not require significant additional time to process or additional pre-processing steps.

## BUILDING THE APPLICATION

Now, we will build the example application. Here, we will implement some data cleaning rules for our datasets. So, first we will create a dataset that contains these rules. In a production application, these rules might be stored in a SQL database, an excel workbook, or whatever is convenient to the users of the application to define the rules.

```
data rules;
  length rule enforce_cond $1024;
  input
    rule_num
    rule_table $
    rule  & $
    enforce_cond & $;
datalines;
1 payments paym_bill eq bills.bill_num  error "PAYMENT has no matching
BILL record"
2 payments paym_source eq payments_source.paym_source_num  error
"PAYMENT has no matching PAYM_SOURCE record"
3 bills    bill_insurance_max ge payments.paym_amount.sum  error
"PAYMENT AMOUNT is more than BILL_INSURANCE_MAX"
4 bills    bill_type eq bill_types.bill_type_num  error "BILL TYPE has
no matching BILL_TYPES record"
;;;;
run;
```

Our rule table has four variables: a rule number, the table the rule applies to, the syntax of the rule itself (*rule*), and what to do if the rule fails (*enforce_cond*).

### SOME HELPER MACROS

We will use a few utility macros in writing this application, mostly to simplify the syntax of our macros. The first quotes a list stored in a macro variable (so *a b c d e* becomes *"a" "b" "c" "d" "e"*) and inserts a separator between them (optional). The second modifies a list by adding a prefix to each element in the list, and again optionally inserts a separator.

```
%macro quoteList(list=,sep=);
  %local _i _count _result;
  %let _result= ;
  %let count = %sysfunc(countw(&list.));
  %do _i = 1 %to &count.;
    %if &_i ne 1 %then %let _result=&_result.&sep.;
    %let _result=&_result.%sysfunc(quote(%scan(&list.,&_i.)));
  %end;
  &_result.
%mend quoteList;

%macro prefixList(list=,prefix=,sep=);
  %local _i _count _result;
  %let _result= ;
  %let count = %sysfunc(countw(&list.));
  %do _i = 1 %to &count.;
    %if &_i ne 1 %then %let _result=&_result.&sep.;
    %let _result=&_result.&prefix.%scan(&list.,&_i.);
  %end;
  &_result.
%mend prefixList;
```

#### DECLARING THE HASH TABLE

The first core application element is the macro that declares the hash table itself. It takes several arguments; the name of the dataset that is loaded into the hash table, the name to assign the hash table itself, the list of key variables, and the list of data variables. These will be passed from the rule table when the macro is called. This also uses the construct `if 0 then set <dataset>;`, which is used to initialize the variables that will comprise the hash table without needing to individually initialize each variable – since `if 0` means the line will never actually be executed, we are simply defining the PDV elements in the compilation stage.

```
%macro defineHash(hashName=, dataset=, keyList=, dataList=);
  if _n_ eq 1 then do;
    if 0 then set &dataset.;
    declare hash &hashName.(dataset:"&dataset.",multidata:'y');
    &hashName..defineKey(%quoteList(list=&keyList.,sep=%str(,)));
    &hashName..defineData(%quoteList(list=&dataList.,sep=%str(,)));
    &hashName..defineDone();
  end;
%mend defineHash;
```

#### PERFORMING A LOOKUP

Next, we will write a simple macro for performing a lookup, using the `find()` method. For this, we will pass the name of the hash, optionally the return code variable, and the list of key values.

```
%macro doHashFind(hash=,rc=rc,key=);
  &rc. = &hash..find(%prefixlist(list=&key.,prefix=key:,sep=%str(,)));
%mend doHashFind;
```

Other modules can easily be written to do things beyond simply find matching rows. For example, here we write a short module that creates a *sum* variable which stores the sum of a value from all matching rows in the hash.

```
%macro doHashSum(hash=,rc=rc,key=,sumVar=, sumOf=);
  &rc. = &hash..find(%prefixlist(list=&key.,prefix=key:,sep=%str(,)));
  do while (&rc. eq 0);
    &sumVar. = sum(&sumVar.,&sumOf.);
    &rc. =
&hash..find_next(%prefixlist(list=&key.,prefix=key:,sep=%str(,)));
  end;
%mend doHashSum;
```

This takes two additional parameters; what variable to store the sum in, and what variable to take the sum of. That variable must be on the `defineData()` statement, of course.

#### IMPLEMENTING THE RULES

Finally, we have two short macros to check and enforce the rules. They both accept two parameters; the rule number, and the logic to define the rule or to enforce it.

```
%macro checkRule(ruleNum=, ruleLogic=);
  rule_&ruleNum. = (&ruleLogic.);
%mend checkRule;

%macro enforceRule(ruleNum=,enforceLogic=);
  if not rule_&ruleNum. then do;
    &enforceLogic.;
  end;
%mend enforceRule;
```

#### A FIRST CLEANING DATA STEP

At this point we have a straightforward data step which we can call like so:

```
data check_payments;
  set payments;
  %defineHash(hashName=bills,dataset=bills,keyList=patient_id bill_num
,dataList=bill_Num);
  %doHashFind(hash=bills,rc=rc_bills,key=patient_id paym_bill);
  %checkRule(ruleNum=1, ruleLogic=%str(paym_bill = bill_num));
  %enforceRule(ruleNum=1,enforceLogic=%str(error "PAYMENT has no
matching BILL record";));
run;
```

This will create a new dataset which validates that every payment has a bill record to match to. The next step is to bring in the rules table to create these automatically.

#### DEFINING THE TABLE RELATIONSHIPS

One more dataset will be needed for this step: one defining the relationships between the tables. SAS has no way to know that on bills, *bill_num* relates to payments' *paym_bill*, after all; so we must define these relationships.

Here we will do that by storing the relationships in a table – similar to the rules table, this might be stored in a production application in a SQL table, an excel worksheet, or similar. It has four fields; the two tables that are being related, and the two sets of keys that are used in that relationship.

```
data table_keys;
  length table_a table_b $32
         keys_a keys_b $512
  ;
  infile datalines truncover;
  input table_a $ table_b $ keys_a $ & keys_b $ &;
datalines;
payments bills patient_id paym_bill  patient_id bill_num
payments payments_source paym_source  paym_source_num
bills bill_types bill_type  bill_type_num
;;;;
run;
```

Then, we will create a format from this table. We might as easily use a hash table or any other combination method; in this case, the format seems easiest as we will be using this in-line, and there is a one to one relationship.

```
data for_fmt_keys;
  length start $65 label $512;
  set table_keys;
  retain fmtname '$tableKeysF' type 'c';
  start = cats(table_a,'|',table_b);
  label = keys_a;
  output;
  start = cats(table_b,'|',table_a);
  label = keys_b;
  output;
  if _n_=1 then do;
    hlo='o';
    start=' ';
    label=' ';
    output;
  end;
run;

proc format cntlin=for_fmt_keys;
quit;
```

**PREPROCESSING THE RULES TABLE**

The next step is to set the rules table up in the manner we need it to be for creating the macro calls. This step will vary some based on how the rules table is defined; in this case, we preferred to define the rules in a very straightforward, nearly-english-language manner which will allow the end user to easily define the rules without knowing much about the application We use dot notation as our one programming conceit, defining the dataset variable as *dataset.variable*, and performing operations such as sum using a third level, *dataset.variable.sum*.

This requires some preprocessing to separate out. We need to do a few things here. First, identify which tables need hash tables defined for them. Second, modify the rule logic to remove the dot notation (as that does not actually work in SAS, of course). Third, identify rules that require summation and split that out as well. We will use regular expressions to do this work in this example, but for users unfamiliar with regular expressions, all of this could be done using traditional SAS tools such as `find()`, `scan()`, `tranwrd()`, and `substr()`.

The first dataset here contains one row per rule, and defines the table, the modified rule logic, and the summation variables.

```
proc sort data=rules;
  by rule_table  rule_num;
run;
data for_calls_tables;
  set rules;
  by rule_table;
  rx_table = prxparse('~([A-Z_][A-Z_0-9]{0,31})\.([A-Z_][A-Z_0-
9]{0,31})~ios');
  rc_table = prxmatch(rx_Table,rule);
  tableMatch = prxposn(rx_table,1,rule);
  varMatch = prxposn(rx_table,2,rule);
  rx_removeTable = prxparse(cats('s~',tableMatch,'\.([A-Z_][A-Z_0-
9]{0,31})~$1~is'));
  rule_notable = prxchange(rx_removeTable,-1,rule);
  rx_Sum = prxparse('s~([A-Z_][A-Z_0-9]{0,31})\.sum~$1_sum~ios');
  rc_sum = prxmatch(rx_Sum,rule_notable);
  if rc_sum then do;
    sum_var    = prxposn(rx_sum,1,rule_notable);
    rule_nosum = prxchange(rx_Sum,-1,rule_notable);
  end;
  else rule_nosum=rule_notable;
run;
```

The second data step contains one row per hash table, and combines the key variables across those rows.

```
proc sort data=for_calls_Tables;
  by rule_table tableMatch;
run;

data final_calls_tables;
  set for_calls_tables;
  by rule_Table tableMatch;
  length varMatchList $1024;
  retain varMatchList;
  if first.tableMatch then do;
    varMatchList=' ' ;
  end;

  if findw(varMatchList,varMatch)=0 then do;
    varMatchList = catx(' ',varMatchList,varMatch);
  end;
  if last.tableMatch then output;
run;
```

These datasets can then be used to create macro calls, like the following.

```
%let table_run = bills;

proc sql;
  select distinct
cats('%defineHash(hashName=',tableMatch,',dataset=',tableMatch,

',keyList=',put(cats(tableMatch,'|',rule_table),$tableKeysF.),
                      ',dataList=',varMatchList,')')
      into :defineHashList separated by ' '
      from final_calls_tables
      where rule_table eq "&table_run";
  select distinct
cats('%doHashFind(hash=',tableMatch,',rc=rc_',tableMatch,

',key=',put(cats(rule_table,'|',tableMatch),$tableKeysF.),')')
      into :doHashFindList separated by ' '
      from final_calls_tables
      where rule_table eq "&table_run";
  select distinct
cats('%doHashSum(hash=',tableMatch,',rc=rc_',tableMatch,

',key=',put(cats(rule_table,'|',tableMatch),$tableKeysF.),
                    ',sumvar=',cats(sum_var,'_sum'),',sumOf=',sum_var,
                  ')')
      into :doHashSumList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run" and not missing(sum_var);

  select
cats('%checkRule(rulenum=',rule_num,',ruleLogic=%str(',rule_nosum,'))')
      into :checkRuleList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run"
      order by rule_num;
  select
cats('%enforceRule(ruleNum=',rule_num,',enforceLogic=%str(',enforce_con
d,'))')
      into :enforceRuleList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run"
      order by rule_num;
  select name
    into :keepList separated by ' '
    from dictionary.columns
    where memname=upcase("&table_run") and libname="WORK";
quit;
```

Six macro variable lists are defined here. `&defineHashList` contains the call(s) to define hash tables; `&doHashFindList` and `&doHashSumList` contain calls to perform lookups and sums, respectively; and `&checkRuleList` and `&enforceRuleList` check and enforce the rules, respectively. Finally, a keep list is constructed from `dictionary.columns` (a SQL table that is effectively identical to the PROC CONTENTS output) which will help us drop unnecessary variables. The keep list should be implemented only in production code, though, as it makes it harder to debug during development and testing. We also have a macro variable up top defining which table we are cleaning; when we wrap this all in a larger macro that will execute per table this will be a parameter to that macro.

Then, we have a simple datastep implementing these macro calls:

```
data &table_run._cleaned;
  set &table_run.;
  &defineHashList.
  &doHashFindList.
  &doHashSumList.
  &checkRuleList.
  &enforceRuleList.
  keep &keepList. rule:;
run;
```

And, voilà, we have our cleaned dataset.

## PUTTING IT ALL TOGETHER – THE FINAL APPLICATION

The final application, then, is a macro that runs the code above once per table defined in the *rules* dataset. Some care should be taken with this macro; in particular, it is helpful to initialize some of the lists to missing in case there are no rows in the *rules* table that qualify (in which case they would cause an error). In the interest of simplicity of presentation, parameter checking is left out here, as is more extensive quoting that would likely be a good idea in a production-quality application.

```
%macro run_rules(table_run=);

%local defineHashList doHashSumList doHashFindList checkRuleList
enforceRuleList keepList;

%let doHashFindList=;
%let doHashSumList=;

proc sql;
select distinct
  cats('%defineHash(hashName=',tableMatch,',dataset=',tableMatch,
  ',keyList=',put(cats(tableMatch,'|',rule_table),$tableKeysF.),
  ',dataList=',varMatchList,')')
      into :defineHashList separated by ' '
      from final_calls_tables
      where rule_table eq "&table_run";
 select distinct
      cats('%doHashFind(hash=',tableMatch,',rc=rc_',tableMatch,
      ',key=',put(cats(rule_table,'|',tableMatch),$tableKeysF.),')')
      into :doHashFindList separated by ' '
      from final_calls_tables
      where rule_table eq "&table_run";
  select distinct
      cats('%doHashSum(hash=',tableMatch,',rc=rc_',tableMatch,
      ',key=',put(cats(rule_table,'|',tableMatch),$tableKeysF.),
      ',sumvar=',cats(sum_var,'_sum'),',sumOf=',sum_var,
      ')')
      into :doHashSumList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run" and not missing(sum_var);

  select cats('%checkRule(rulenum=',rule_num,',ruleLogic=%str(',
      rule_nosum,'))')
      into :checkRuleList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run"
      order by rule_num;
```

```
  select cats('%enforceRule(ruleNum=',rule_num,',enforceLogic=%str(',
      enforce_cond,'))')
      into :enforceRuleList separated by ' '
      from for_calls_tables
      where rule_table eq "&table_run"
      order by rule_num;
  select name
      into :keepList separated by ' '
      from dictionary.columns
      where memname=upcase("&table_run") and libname="WORK";
quit;

data &table_run._cleaned;
  set &table_run.;
  &defineHashList.
  &doHashFindList.
  &doHashSumList.
  &checkRuleList.
  &enforceRuleList.
  keep &keepList. rule:;
run;
%mend run_rules;

proc sql;
  select distinct cats('%run_rules(table_run=',rule_table,')')
    into :runRulesList separated by ' '
    from rules;
quit;

&runRulesList.
```

## ROOM FOR GROWTH

There are a fair number of different concepts that could be incorporated into this, depending on the needs of the project. Simple modules could be added, such as *max* and *min* modules (obtaining the max and min from a variable of all matching rows). A fairly simple modification would enable this to work with three or more tables in one rule. Enforcement could involve modifying the data rather than simply defining a log message.

Moreover, other applications beyond validation/cleaning could be designed using this: for example, this could be used to perform reporting tasks just as easily – the current macro, after all, is capable of summations at effectively a class level not all that different from a simple version of PROC MEANS.

## CONCLUSION

Faster than merging or SQL joins, more flexible than formats, and easier to implement than temporary arrays, hash tables are a powerful tool in the Data-Driven Programmer's tool belt for developing applications that are flexible, data-driven, and easy to use for non-programmer users, while maintaining efficiency.

## REFERENCES AND RECOMMENDED READING

Dorfman, Paul et al, 2005. "Data Step Hash Objects as Programming Tools." *Proceedings of SUGI 30*. Cary, North Carolina; SAS Institute, Inc. Available at http://www2.sas.com/proceedings/sugi30/236-30.pdf (A good hash tutorial.)

Fehd, Ronald and Art Carpenter, 2007. "List Processing Basics: Creating and Using Lists of Macro Variables". *Proceedings of SAS Global Forum 2007.* Cary, North Carolina; SAS Institute, Inc. Available at http://www2.sas.com/proceedings/forum2007/113-2007.pdf (Covers various techniques for pulling data into macro variables.)

## ACKNOWLEDGMENTS

Thank you to Jeff Vose and David Trevarthen at NORC for making it possible to attend this conference and encouraging me to write this paper. Additionally, thanks to Marcus Maher whose use of hash tables in reporting programs encouraged me to think of this topic and to use this process in my own work. Finally, thanks to Paul Dorfman, without whom SAS hash tables would likely not exist today, and for sharing many techniques over the years on SAS-L which helped inform my work and ultimately this paper.

## CONTACT INFORMATION

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

Joe Matise
NORC at the University of Chicago
55 E Monroe
Chicago, IL 60603
Matise.joe@gmail.com