SAS418-2017

Dictionaries: Referencing a New PROC FCMP Data Type

Andrew Henrick, Mike Whitcher, and Karen Croft, SAS Institute Inc.

ABSTRACT

Hash objects have been supported in the DATA step and the FCMP procedure for a while, but have you ever felt that hash objects could do a little more? For example, what if you needed to store more than doubles and character strings? Introducing PROC FCMP dictionaries. Dictionaries allow you to create references not only to numeric and character data, but they also give you fast in-memory hashing to arrays, other dictionaries, and even PROC FCMP hash objects. This paper gets you started using PROC FCMP dictionaries, describes usage syntax, and explores new programming patterns that are now available to your PROC FCMP programs, functions, and subroutines in the new SAS® Viya™ platform environment.

INTRODUCTION

In computer science, hashing refers to a search algorithm. Through the use of a hash function, a collection of input strings and/or numbers (*key*) are converted to a single value, usually a number (*hash)*. This hash value is then used as an index into a *hash table*. Each key thus provides a means to store and retrieve associated data within the hash table. Hash tables are considered one of the most efficient ways to search large amounts of data.

The FCMP procedure enables you to create, test, and store SAS® functions and CALL routines. The FCMP hash object has given users the ability to define and utilize hash tables within their FCMP programs since SAS® 9.3. In the SAS® Viya™ 3.3 release FCMP users will have access to a new tool, the dictionary. Dictionaries also store key/value pairs. To experienced users of the FCMP hash object there are similarities. However, dictionaries are more flexible in not only what types of data can be stored but in how they are stored.

FCMP HASH OBJECTS VS DICTIONARIES

Since hashing has been available to PROC FCMP users for over five years, some programmers may be quite familiar with its usefulness. Dictionaries also provide hashing within PROC FCMP functions and subroutines, but how should a programmer go about choosing one over the other?

A SIMPLE EXAMPLE

Let's start with a sample program within PROC FCMP that demonstrates how to use a hash object. This program is too simple to demonstrate the strengths of hashing, but its brevity allows it to be included in its entirety:

```
proc fcmp;
  declare hash h;

  rc = h.definekey('x');
  rc = h.definedata('y');
  rc = h.definedone();

  do x = 1 to 10;
     y = x**2;
     rc = h.add();
  end;

  x = 4;
  rc = h.find();
  put x= y=;
```

```
run;
```

Now how would we accomplish the same thing with dictionaries? Here's the rewritten program:

```
proc fcmp;
  declare dictionary d;
  do x = 1 to 10;
     d[x] = x**2;
  end;

  x = 4;
  y = d[x];
  put x= y=;
run;
```

```
x=4 y=16
```

Output 1. Output from Simple Example

The syntax and usage of insert and find operations with dictionaries works just like an array. You could modify the program to use an array of ten elements instead of a dictionary. You would get better performance with an array since indexing is simple and fast.

Hashing allows you to use character strings as keys, so here's an example of a dictionary that cannot be replaced by an array:

```
proc fcmp;
  array keys[5] $ ("alpha" "bravo" "charlie" "delta" "echo");
  declare dnary d;
  do x = 1 to 5;
     d[x]= x;
     d[keys[x]] = 2*x;
     d[x,keys[x]] = x**2;
  end;

  x = d[4];
  y = d[keys[x]];
  z = d[x, keys[x]];
  put x= keys[x]= y= z=;
run;
```

```
x=4 keys[4]=delta y=8 z=16
```

Output 2. Output from Character Key Example

Notice the abbreviation of the keyword "dictionary" as "dnary". This has been provided as a convenience. Also notice that the dictionary allows for a variable number of keys (up to six). Even though the syntax is similar to array indexing, this doesn't correspond to the dimensions of an array. Once you define an array as two dimensional, you have to provide two (and only two) indices.

REASONS TO CHOOSE AN FCMP HASH OBJECT

There is a great deal of flexibility in the options provided when declaring a hash object. For example, you can fill the hash object using a data set, the contents can be ordered (ascending or descending), and you can setup multiple iterators for the same data. There's no limit on the number of keys. However, there are a few constraints. All key and data variables must be predefined. Once the hash object has been

setup, its configuration cannot be changed. Data must be either numeric or character and is stored by value within the hash object.

Discussing FCMP hash and hash iterator (hiter) objects fully is beyond the scope of this paper. If you wish to know more about hash objects, please refer to the FCMP Procedure section of the *SAS® Viya™ 3.3 Visual Data Management and Utility Procedures Guide*. In the References and Recommend Reading sections you can also find papers and books dedicated to the topic.

REASONS TO CHOOSE A DICTIONARY

Dictionaries support a maximum of six numeric or character keys. Keys do not have to be all character or all numeric; they can be mixed. Data is not limited to just numeric or character data, but you can store other container types. Arrays, hash objects, and dictionaries themselves are valid data members within a dictionary. For performance reasons these data types are stored by reference. For arrays, it is an option to have the dictionary store its contents by value, but deep copies of hash objects and dictionaries are not permitted.

Similarly, numeric and character data is stored by value, but there is an option to store numeric or character variables within a dictionary by reference. For more information, please refer to the helper functions outlined in Appendix A. FCMP Dictionary Syntax.

What is to be stored in a dictionary does not have to be predefined with a DEFINEDATA call as you would do for hash objects. It is permitted to change the type of what is stored at a given key position. Adding data to a dictionary for an existing key is always considered a replace action.

ITERATING WITHIN DICTIONARIES

Maintaining a separate iterator object is not necessary with a dictionary; it provides its own iterator behavior. Given the variability of the types of data that can be stored within a dictionary, iterating has its own challenges.

STARTING SIMPLE

Let's start with a program that walks a dictionary filled with numeric doubles just like the one in our first example. Here is the code:

```
proc fcmp;
   declare dnary d;

   do x = 1  to 10;
      d[x] = x**2;
   end;

   rc = d.first(y);
   if (rc eq 0) then put y=;
   y = d.next();
   do while (MISSING(y) ne 1);
      put y=;
      y = d.next();
   end;
run;
```

Much like hash objects, dictionaries have a set of built-in helper functions. In the above example we have used FIRST and NEXT to step through the dictionary and display each element. FIRST initializes the iterator and if successful, places the first data in the data argument 'y'. Then NEXT is called to advance the iterator and return the next double. Since we know we have set up the dictionary with non-missing values, we can test for a returned missing value from the NEXT function to determine when we've reached the end of the dictionary. A later example will show you the right way to test for the last element so you don't have to depend on tests for missing values.

```
y=4
y=9
y=16
y=25
y=36
y=49
y=64
y=81
y=100
y=1
```

Output 3. Output from First Iterator Example

As you can see from the output, key/value pairs are not ordered. While you will get replicable results by inserting the same data in the same order, this isn't guaranteed behavior and may change in the future. If you wish to traverse the dictionary in reverse, you can replace the calls to FIRST and NEXT with LAST and PREV respectively. The program can be simplified even further as:

```
proc fcmp;
   declare dnary d;

   do x = 1  to 10;
      d[x] = x**2;
   end;

   y = d.next();
   do while (MISSING(y) ne 1);
      put y=;
      y = d.next();
   end;
run;
```

If the iterator is uninitialized, NEXT returns the first data item. We can safely remove the call to FIRST and the corresponding check of its return code. Now let's say we want to iterate across a dictionary with multiple types of data and only pick out the doubles? Here's the code:

```
proc fcmp;
  declare dnary d;
  array a[10];

  /* first fill the dictionary with numeric elements */
  do x = 1 to 10;
     a[x] = x**2;
     d[x] = x**2;
  end;

  /* now replace a few entries with strings and an array */
  d[4] = "alpha";
  d[8] = "bravo";
  d[9] = a;

  /* give the iterator its starting point with FIRST or NEXT */
```

```
  rc = d.first(y, type, is_array);

  /* check for a double and initialize sum_d accordingly */
  if (type eq 1) and (is_array eq 0) then sum_d = y;
  else sum_d = 0;

  /* use HASNEXT to test for end of dictionary */
  do while (d.hasnext(type, is_array) eq 1);
     /* only add doubles to our sum */
     if (type eq 1) and (is_array eq 0) then do;
        y = d.next();
        sum_d += y;
     end;
     else
        rc = d.skipnext();
  end;

  /* sum up array elements that match double entries in dictionary */
  sum_a = a[1] + a[2] + a[3] + a[5] + a[6] + a[7] + a[10];
  put sum_d= sum_a=;
run;
```

For comparison, an array is filled with the same values as the dictionary and a sum is done on the dictionary's numeric indices.

```
sum_d=224 sum_a=224
```

Output 4. Output from Multiple Types Example

The dictionary is filled with just doubles as before but then three elements are replaced with strings and an array. The iterator function FIRST is used to get things started but then HASNEXT is used to see if we've reached the end of the dictionary. For each data item, the dictionary returns some extra information using the optional arguments 'type' and 'is_array'. This way we know when we've found a double and not a numeric array. When HASNEXT reports the next item is not a double, SKIPNEXT advances the iterator to the next position.

Data Type	Numeric Code
double	1
char	2
varchar	3
dictionary	-1
hash object	-2
hash iterator	-3

Table 1. Possible Return Values for 'Type'

Again if you wish to traverse the dictionary in reverse, iterator functions LAST, HASPREV, PREV, and SKIPPREV are available. For more on dictionary helper functions and their use, please see Appendix A. FCMP Dictionary Syntax.

DICTIONARIES AS FUNCTION ARGUMENTS

Giving the user the ability to break down a program into reusable blocks of code is the primary function of PROC FCMP. Given the breadth and scope of what can be stored in a dictionary, limiting the user to declaring and working with a dictionary only within a single function or subroutine was considered far too

constrictive. To remove this restriction, dictionaries have been added as valid PROC FCMP function arguments.

For this example let's say we have a group of functions in a single package that do the following:

```
proc fcmp outlib=work.myfuncs.mypkg;
   function prod(x, y, z);
      return(x * y * z);
   endsub;

   function const(x $);
      return(constant(x));
   endsub;

   function sum_arr(x[*]);
      sum = 0;
        do i = 1 to dim(x);
           sum += x[i];
        end;
        return(sum);
   endsub;
run;
```

Each function does something different based on the arguments passed, but the operation and number of arguments is based on the type of the first argument. This presents an opportunity to create a single function that maps dictionary data to a specific function. All we need to do is create a utility function that ties the two together:

```
/* set option CMPLIB so that proc fcmp will load our function library */
option CMPLIB=work.myfuncs;

proc fcmp;
   function get_value(d dnary);
      length str $ 8;
      array  arr[10];

      /* get the datatype and array settings for a specific key's data */
      type = d.describe(is_array, "x");

      /* using the datatype we can determine which function to call */
      select (type);
           when(1) do;
              /* doubles, scalars first then the array case */
              if (is_array eq 0) then
                 result = prod(d["x"], d["y"], d["z"]);
              else do;
                 arr = d["x"];
                 result = sum_arr(arr);
              end;
           end;
           when(2) do;
              /* character strings */
              str = d["x"];
              result = const(str);
           end;
           otherwise result = 0;
      end;
      return(result);
```

```
   endsub;

   declare dnary d;
   d["x"] = 5;
   d["y"] = 6;
   d["z"] = 7;
   num_res = get_value(d);

   array arr[10];
   do x = 1 to 10;
      arr[x] = x**2;
   end;

   /* replace the 5 with an array and call again */
   d["x"] = arr;
   arr_res = get_value(d);

   /* replace the array with a string literal and call again */
   d["x"] = "pi";
   char_res = get_value(d);

   put num_res= arr_res= char_res=;
run;
```

```
num_res=210 arr_res=385 char_res=3.1415926536
```

Output 5. Output from Dictionary Function Argument Example

Notice the need to retrieve dictionary data and store it in a local variable. For numeric temps, this isn't necessary such as in the call to the function 'prod'. However with complex types like hash and array, there currently is no array temp or hash table temp, so we need a local variable. Character temps are permissible, so why then is the local variable 'str' necessary? The problem is when function get_value() is compiled, what is stored in d["x"] is ambiguous. We know that 'const' expects a character argument but the return value of a dictionary find operation has to have a default, so it defaults to numeric. By assigning d["x"] to a local variable, the find operation can handle mismatches when running the program. This is a common problem; type conflicts that would normally be caught at compile time can become nebulous as data is stored/retrieved from a dictionary. This necessitates postponing type checking and error reporting until run time.

Lastly it is important to point out that dictionary arguments are passed by reference. This can be useful as you can edit the contents of the dictionary and then see those changes outside of the function. However since dictionaries themselves can store references to local variables within the function, this is undefined behavior that will result in errors. When modifying dictionary arguments within functions, it is important to store those changes by value. For numeric and character data this is the default; do not use the REF function. Let's look at an example of the wrong way to store an array:

```
proc fcmp;
   function store_arr(d dnary, key);
      array arr[10];

      do x = 1 to 10;
         arr[x] = x**2;
      end;

      /* danger! Ok inside the function but not outside */
      d[key] = arr;
```

```
      return(0);
   endsub;

   declare dnary d;
   array x[10];

   rc = store_arr(d, 5);

   /* access the stored array, this is undefined and may crash */
   x = d[5];
   put x=;
run;
```

To fix the above program, use the CLONE helper function to perform a deep copy of the array data:

```
proc fcmp;
   function store_arr(d dnary, key);
      array arr[10];

      do x = 1 to 10;
         arr[x] = x**2;
      end;

      /* much better, the dictionary will perform a full copy of 'arr' */
      rc = d.clone(arr, key);
      return(rc);
   endsub;

   declare dnary d;
   array x[10];

   rc = store_arr(d, 5);

   /* access the stored array */
   if rc eq 0 then x = d[5];
   put x=;
run;
```

```
x[1]=1 x[2]=4 x[3]=9 x[4]=16 x[5]=25 x[6]=36 x[7]=49 x[8]=64 x[9]=81
x[10]=100
```

Output 6. Output from Storing an Array by Value Example

Hash objects, hash iterators, and dictionaries cannot be stored by value.

CONCLUSION

Dictionaries provide a versatile and fast utility for use within PROC FCMP functions and subroutines. In this overview of dictionary syntax and their helper functions, we hope that our users discover another useful tool available to them in the upcoming SAS® Viya™ 3.3 release.

REFERENCES

Dorfman, Paul, Shajenko, Lessia,and Vyverman, Koen. 2008. "Hash Crash and Beyond", *Proceedings of the SAS Global Forum 2008 Conference* - http://www2.sas.com/proceedings/forum2008/037-2008.pdf

Loren, Judy. 2008. "*How* Do I Love Hash Tables? Let Me Count The Ways*!*", *Proceedings of the SAS Global Forum 2008 Conference* - http://www2.sas.com/proceedings/forum2008/029-2008.pdf

Henrick, A., D. Erdman, and S. Christian. 2013. "Hashing in PROC FCMP to Enhance Your Productivity." *Proceedings of the SAS Global Forum 2013 Conference,* Cary, NC. SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings13/129-2013.pdf

SAS Institute Inc. 2016. "The FCMP Procedure." *Base SAS® 9.4 Procedures Guide. 6th edn.* Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/proc/69850/PDF/default/proc.pdf

SAS Institute Inc. 2016. SAS® 9.4 *Component Objects: Reference*. Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/lecompobjref/69740/PDF/default/lecompobjref.pdf

RECOMMENDED READING

- *SAS® Viya™ 3.3 Visual Data Management and Utility Procedures Guide*
- *SAS® Hash Object Programming Made Easy*

CONTACT INFORMATION

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

Andrew Henrick
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-3362
Andrew.Henrick@sas.com

Mike Whitcher
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-7936
Mike.Whitcher@sas.com

Karen Croft
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-3912
Karen.Croft@sas.com

APPENDIX A. FCMP DICTIONARY SYNTAX

With the SAS® Viya™ 3.3 release, the following statements and methods are supported for FCMP dictionaries:

- DECLARE
- CLONE
- REF
- DESCRIBE
- REMOVE
- CLEAR
- NUM_ITEMS
- FIRST
- LAST
- NEXT
- PREV
- HASNEXT
- HASPREV
- SKIPNEXT
- SKIPPREV

DECLARE STATEMENT

The DECLARE statement is used to create a new instance of a dictionary. The syntax for the DECLARE statement is as follows:

```
DECLARE dictionary object-name;
```

The DECLARE statement along with the 'dictionary' or 'dnary' keyword tells PROC FCMP that you wish to create a new dictionary object with the name provided.

CLONE METHOD

For performance reasons, complex data types like arrays, hash objects, and dictionaries are stored within a dictionary by reference. The CLONE method gives you the ability to store an array by value. Dictionaries do not support storing hash objects, hash iterators, and dictionaries by value. Numeric and character data is stored by value by default, so use of the CLONE method is supported but redundant.

```
rc = object.CLONE(data,key-1<,…key-n>);
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

key-1…key-n

The key values indicating where 'data' is to be placed. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

data

The data you wish to pair with the keys specified. The method will attempt to store this data by value.

REF METHOD

By default, numeric and character data is stored by value. The REF method gives you the ability to store numeric and character variables by reference.

```
rc = object.REF(data,key-1<,…key-n>);
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

key-1…key-n

The key values indicating where the reference to 'data' is to be placed. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

data

The data you wish to pair with the keys specified. The method will store this data by reference.

DESCRIBE METHOD

Provides information about what is stored at a particular location within the dictionary. This enables you to programmatically determine what variables to use when retrieving data from a dictionary.

```
datatype = object.DESCRIBE(is_array,key-1<,…key-n>);
```

Arguments

datatype

Specifies the type of the data found as a numeric value or MISSING if there is no data.

object

The name of the dictionary defined in the DECLARE statement.

is_array

Set to 1 if the data found is an array, set to zero if not. This value will also be set to MISSING if there is no data.

key-1…key-n

The key values indicating the data location to describe. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

Data Type	Numeric Code
double	1
char	2
varchar	3
dictionary	-1
hash object	-2
hash iterator	-3

Table 2. Possible Return Values for 'Datatype'

REMOVE METHOD

Removes a key/value pair from the dictionary. The number of items in the dictionary is effectively reduced by one.

```
rc = object.REMOVE(key-1<,…key-n>);
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

key-1…key-n

The key values indicating the data location to describe. Keys can be expressed as numeric or character literals or variables. At least one key must be provided.

CLEAR METHOD

Removes all key/value pairs from the dictionary.

```
rc = object.CLEAR();
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

NUM_ITEMS METHOD

Returns the number of items in a dictionary.

```
variable_name = object.NUM_ITEMS();
```

Arguments

variable_name

The number of items in the dictionary is returned, not a return code, so direct assignment to a local numeric variable is expected.

object

The name of the dictionary defined in the DECLARE statement.

FIRST METHOD

Updates a provided variable with the first data item in the dictionary. Also returns information about the found data item if desired.

```
rc = object.FIRST(datavar<,datatype,is_array>);
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

datavar

Location to store a copy of the first data item. If the provided data variable is of the wrong type and incompatible, it will be set to MISSING and an error will be seen.

datatype

Specifies the type of the data found as a numeric value or MISSING if there is no data. Please see the DESCRIBE method for details about what values correspond to each possible data type.

is_array

Set to 1 if the data found is an array, set to zero if not. This value will also be set to MISSING if there is no data.

LAST METHOD

Updates a provided variable with the last data item in the dictionary. Also returns information about the found data item if desired.

```
rc = object.LAST(datavar<,datatype,is_array>);
```

Arguments

rc

Specifies if the method succeeded so that the return code can be checked within the FCMP function code. A return value of 0 indicates success.

object

The name of the dictionary defined in the DECLARE statement.

datavar

Location to store a copy of the last data item. If the provided data variable is of the wrong type and incompatible, it will be set to missing and an error will be seen.

datatype

Specifies the type of the data found as a numeric value or MISSING if there is no data. Please see the DESCRIBE method for details about what values correspond to each possible data type.

is_array

Set to 1 if the data found is an array, set to zero if not. This value will also be set to MISSING if there is no data.

NEXT METHOD

Returns the next data item in the dictionary. The first data item will be returned if the iterator is uninitialized.

```
datavar = object.NEXT();
```

Arguments

datavar

Location to store a copy of the next data item. If the provided data variable is of the wrong type and incompatible, it will be set to missing and an error will be seen.

object

The name of the dictionary defined in the DECLARE statement.

PREV METHOD

Returns the previous data item in the dictionary. The last data item will be returned if the iterator is uninitialized.

```
datavar = object.PREV();
```

Arguments

datavar

Location to store a copy of the previous data item. If the provided data variable is of the wrong type and incompatible, it will be set to missing and an error will be seen.

object

The name of the dictionary defined in the DECLARE statement.

HASNEXT METHOD

Returns whether or not there is a next data item in the dictionary. Also returns information about the found data item if desired. If the iterator is uninitialized, HASNEXT will describe the first data item in the dictionary.

```
found = object.HASNEXT(<datatype,is_array>);
```

Arguments

found

Set to 1 if the dictionary's iterator has a next data item, 0 if not.

object

The name of the dictionary defined in the DECLARE statement.

datatype

Specifies the type of the data found as a numeric value or MISSING if there is no data. Please see the DESCRIBE method for details about what values correspond to each possible data type.

is_array

Set to 1 if the data found is an array, set to zero if not. This value will also be set to MISSING if there is no data.

HASPREV METHOD

Returns whether or not there is a previous data item in the dictionary. Also returns information about the found data item if desired. If the iterator is uninitialized, HASPREV will describe the last data item in the dictionary.

```
found = object.HASPREV(<datatype,is_array>);
```

Arguments

found

Set to 1 if the dictionary's iterator has a previous data item, 0 if not.

object

The name of the dictionary defined in the DECLARE statement.

datatype

Specifies the type of the data found as a numeric value or MISSING if there is no data. Please see the DESCRIBE method for details about what values correspond to each possible data type.

is_array

Set to 1 if the data found is an array, set to zero if not. This value will also be set to MISSING if there is no data.

SKIPNEXT METHOD

Advances the iterator to the next position.

```
found = object.SKIPNEXT();
```

Arguments

found

Set to 0 if the end of the dictionary has been reached, set to 1 otherwise.

object

The name of the dictionary defined in the DECLARE statement.

SKIPPREV METHOD

Moves the iterator over the previous key/value pair in the dictionary.

```
found = object.SKIPPREV();
```

Arguments

found

Set to 0 if the end of the dictionary has been reached, set to 1 otherwise.

object

The name of the dictionary defined in the DECLARE statement.