

The DATA step in SAS 9: What's New?

Jason Secosky, SAS, Cary, NC.

Revised May 3, 2004

INTRODUCTION

This paper presents DATA step language enhancements in SAS 9.0 and 9.1. These enhancements include: Perl regular expressions for fast search and replace in text, hash tables for searching an expandable collection of values based on a key, and sorting values with a DATA step function. Performance improvements that speed existing code are discussed along with new ways to output log messages and retrieve variable values.

PERL REGULAR EXPRESSIONS

Perl Regular Expressions (regex) are an easy and powerful search, replace, and extraction language for text. Traditionally, the INDEX, SCAN, and SUBSTR functions, along with concatenation are used for simple search and replace operations in text. These functions often require many steps that can be error prone and make searching dynamic text difficult, if not impossible. Regex combine several operations into a single function call, making code less error prone, easier to maintain, clearer, and may improve performance. The following sections discuss common cases for using regex in data validation, replacing text, and extracting text.

INTRODUCTION

Regex are a language for searching text that consists of a string of literal characters and special characters called metacharacters. When searching with a regex, a literal character in the regex matches that character in the string being searched. For example, the regex `/abc/` will search for the characters `abc` in a string.

All regex begin and end with a delimiter. In the case of `/abc/`, the delimiter is a forward slash, `/`. The delimiter can be any non-alphanumeric character or matching braces. Delimiters are necessary because blanks in a regex match blanks in a string, so the `/` to make regex bounds clear a delimiter is needed.

The power of regex lies with the metacharacters. The regex metacharacters perform special actions when searching, like forcing the match to begin at a particular location or by matching a particular set of characters. For instance, the regex `/abc\d/` will match `abc` followed by a digit. The metacharacter for a digit is `\d`. Other metacharacters will be introduced throughout the paper and are discussed in the SAS 9 Language Reference Dictionary, at <http://regex.info>, or in [2].

DATA VALIDATION

Data validation involves checking a value to ensure that it matches a particular specification. The following example will validate phone numbers to ensure that there exists a three digit parenthesized area code followed by an optional space, then seven digits with a hyphen between the third and fourth digits. For example, (919) 677-8000 would be a valid phone number, while 919-677-8000 would not because the area code isn't parenthesized.

The functions PRXPARSE and PRXMATCH prepare a regex for use by SAS and perform a search with a regex. PRXPARSE takes a regex and returns a pattern identification number for the compiled regex. In the example below, the identification number, `re`, is retained since the regex does not change and does not need to be compiled for each iteration of the DATA step. PRXMATCH takes a pattern identification number and a character value and returns the position where the regex finds a match, or

zero if no match was found.

```
data _null_;
  length first last phone $ 16;
  retain re;
  if _N_ = 1 then do;
    re = prxparse("/\([2-9]\d\d\) ?" ||
                "[2-9]\d\d-\d\d\d\d/");
  end;

  input first last phone & $16.;

  if ^prxmatch(re, phone) then
    putlog "NOTE: Invalid, "
          first last phone;

datalines;
Thomas Archer      (919)319-1677
Lucy Mallory       800-899-2164
Tom Joad           (508) 852-2146
Laurie Jorgensen   (252)152-7583
;
```

Output:

```
NOTE: Invalid, Lucy Mallory 800-899-2164
NOTE: Invalid, Laurie Jorgensen (252)152-7583
```

The regex passed to PRXPARSE contains the metacharacters `\(` to match an open parenthesis, `\d` a digit, `[2-9]` match the digits from 2 to 9, and `?` matches the prior character one or zero times. In this case, the prior character is a blank, so zero or one blank will be matched. This example shows that with few operations a complex search can be performed with a regex.

DATA VALIDATION – SAS 9.1

In SAS 9.1, using PRXMATCH is simplified by removing the need to call PRXPARSE to compile a regex. Instead, either a regex or pattern identification number can be passed to PRXMATCH.

Passing a regex to PRXMATCH causes PRXMATCH to compile the regex internally before performing a search. To improve performance, when the regex is a constant or an "o" modifier is placed after the last delimiter of the regex, the regex is compiled once during the first call to PRXMATCH. Then, the compiled regex is reused with successive calls to PRXMATCH. The next example simplifies phone number validation by passing a regex to PRXMATCH.

```
data _null_;
  length first last phone $ 16;
  input first last phone & $16.;
  if ^prxmatch("/\([2-9]\d\d\) ?" ||
              "[2-9]\d\d-\d\d\d\d/",
              phone) then
    putlog "NOTE: Invalid, "
          first last phone;

datalines;
Thomas Archer      (919)319-1677
Lucy Mallory       800-899-2164
Tom Joad           (508) 852-2146
Laurie Jorgensen   (252)152-7583
;
```

Passing a regex to PRXMATCH enables PROC SQL queries to use regex. The following example uses PROC SQL to report invalid phone numbers from a table of phone numbers.

```
proc sql;
  select * from phone_numbers
  where
  not prxmatch("/\[([2-9]\d\d\ ) ?" ||
              "[2-9]\d\d-\d\d\d\d\d/", phone);
quit;
```

Using PRXMATCH from a WHERE clause is similar to using the LIKE operator. However, regexp have more specific metacharacters. For instance, validating phone numbers cannot be done with the LIKE operator since there is no metacharacter to search for a digit. Instead, SUBSTR and character comparisons are used to validate each character of a phone number. This type of WHERE clause is tedious to program and has a greater execution time.

REPLACING TEXT

Regex enable easy search and replace operations. These are done by creating a regexp to find a match and by providing a value to replace the match with. PRXPARSE compiles the regexp, but instead of using PRXMATCH to find a match, PRXCHANGE is used to find a match and perform the replacement. PRXCHANGE takes a pattern identification number, the number of times to perform the search and replace, and a character variable to perform the replacement on.

The regexp for a replacement must contain a regexp to search with and replacement text for matches found. The format for this type of regexp is `s/regexp/replacement-text/`. The "s" before the regexp signifies that this is a substitution regexp. The following example replaces all occurrences of a less than sign with `<`, a common substitution when converting text to HTML.

```
data _null_;
  retain re;
  if _N_ = 1 then do;
    re = prxparse('s/</&lt;');
  end;

  input;
  call prxchange(re, -1, _infile_);
  put _infile_;

datalines;
x + y < 15
x < 10 < y
y < 11
;
```

Output:

```
x + y &lt; 15
x &lt; 10 &lt; y
y &lt; 11
```

Notice -1 is passed to PRXCHANGE for the number of times to perform the replacement. -1 is a special value that indicates that all possible replacements should occur.

In a traditional DATA step, search and replace requires multiple function calls and concatenations. With a regexp, the substitution is performed by one function.

REPLACING TEXT – SAS 9.1

In SAS 9.1, using PRXCHANGE is simplified by removing the need to call PRXPARSE to compile a regexp. To pass a regexp to PRXCHANGE, the PRXCHANGE function is used instead of CALL PRXCHANGE. The PRXCHANGE function takes a regexp, the number of times to perform the search and replace, and an input string. This function returns the input string after performing all requested search and replace operations.

Similar to PRXMATCH, PRXCHANGE compiles the regexp

internally during the first call and, if possible, reuses the compiled regexp during successive calls. The next example performs the same search and replace as the prior example, but with fewer lines of code.

```
data _null_;
  input;
  _infile_ = prxchange('s/</&lt;/',
                    -1, _infile_);

  put _infile_;

datalines;
x + y < 15
x < 10 < y
y < 11
;
```

The ability to pass a regexp to PRXCHANGE and return a result enables calling PRXCHANGE from a PROC SQL query. The following query produces a column with the same textural change as the prior example. The query reads from the input table `text_lines`, changes the text for the column `line`, and places the results in a column named `html_line`.

```
proc sql;
  select prxchange('s/</&lt;/', -1, line)
  as html_line
  from text_lines;
quit;
```

Removing the need to call PRXPARSE further simplifies using regexp and enables using regexp outside of the DATA step.

EXTRACTING TEXT

Parentheses are metacharacters in a regexp that enable extracting part of a regexp match by calling PRXPOSN. A parenthesized subexpression enables retrieving the position and length of a part of the total match.

For example, the regexp `/(\d\d\d)-(\d\d\d)/` will match the text 123-456. The two sets of parentheses enable retrieving the position and length of parts or submatches of the total match. In this case, submatch #1 is 123 and submatch #2 is 456. Calling PRXPOSN with a submatch number retrieves the position and length of a submatch.

The following example extracts the area code of a phone number and checks to see if the area code is located in North Carolina. The area code part of the regexp is surrounded by parentheses, in bold, to indicate the area code digits will be submatch #1. In this case, we pass 1 to PRXPOSN to retrieve the position and length of submatch #1, the area code digits.

```
data _null_;
  length first last phone $ 16;
  retain re;
  if _N_ = 1 then do;
    re = prxparse("/\[([2-9]\d\d\ ) ?" ||
                "[2-9]\d\d-\d\d\d\d\d/");
  end;

  input first last phone & $16.;

  if prxmatch(re, phone) then do;
    call prxposn(re, 1, pos, len);
    area_code = substr(phone, pos, len);
    if area_code ^in ("828" "336"
                    "704" "910"
                    "919" "252") then
      putlog "NOTE: Not in N. Carolina: "
            first last phone;
  end;
```

```

datalines;
Thomas Archer      (919)319-1677
Lucy Mallory       (800)899-2164
Tom Joad           (508) 852-2146
Laurie Jorgensen  (252)352-7583
;

```

Output:

```

NOTE: Not in NC, Lucy Mallory (800)899-2164
NOTE: Not in NC, Tom Joad (508) 852-2146

```

EXTRACTING TEXT – SAS 9.1

In SAS 9.1 using PRXPOSN is simplified by not needing to call SUBSTR to retrieve the text for a submatch. The PRXPOSN function, as opposed to CALL PRXPOSN, is passed the original search text instead of position and length variables. The function returns the submatch text. Below is the area code extraction program written with the PRXPOSN function. The new line is in bold.

```

data _null_;
  length first last phone $ 16;
  retain re;
  if _N_ = 1 then do;
    re = prxparse("/\(([2-9]\d\d)\) ?" ||
                "[2-9]\d\d-\d\d\d\d/");
  end;

  input first last phone & $16.;

  if prxmatch(re, phone) then do;
    area_code = prxposn(re, 1, phone);
    if area_code ^in ("828" "336"
                    "704" "910"
                    "919" "252") then
      putlog "NOTE: Not in N. Carolina: "
            first last phone;
  end;
datalines;
Thomas Archer      (919)319-1677
Lucy Mallory       (800)899-2164
Tom Joad           (508) 852-2146
Laurie Jorgensen  (252)352-7583
;

```

To use PRXPOSN, a pattern identification number must be passed to PRXMATCH, as opposed to the simpler form of passing a regexp. After calling PRXMATCH, the positions and lengths of submatches are associated with the pattern identification number. The positions and lengths are retrieved by calling PRXPOSN. This is a situation where the simpler form of passing a regexp to PRXMATCH cannot be used.

PRX FUNCTIONS

A few of the PRX functions have been presented, others include:

- CALL PRXSUBSTR returns the position and length of an entire match.
- CALL PRXNEXT is used to iterate over several regexp matches in text.
- PRXPAREN returns the last submatch that was found.
- CALL PRXFREE frees memory used by a compiled regexp.
- CALL PRXDEBUG toggles output of Perl debug information to the SAS log.

These function are described in the SAS OnlineDoc.

COMPARISON WITH PERL AND RX

The PRX functions are built from the same source code as Perl 5.6.1. This means performance and functionality of the PRX

functions is similar to that of Perl. Differences between Perl and PRX syntax occur where Perl language elements are used within a regexp. For instance, Perl variables and statements are not allowed in a regexp since the regexp is operating in the context of SAS, not Perl.

The SAS RX functions perform similar actions to the PRX functions but use a different regexp syntax. The RX syntax is consistent with other regexp syntax in SAS and provide the ability to score matches, match balanced symbols, and have a few more operations when replacing text. The RX functions are typically slower than the PRX functions and require more steps to perform text extraction. The RX functions cannot be used from PROC SQL or a WHERE clause.

For each of the examples, equivalent code could be written that does not use a regexp. The non-regexp code would involve more function calls, managing character positions in text, and manipulating pieces of text. Regexp combine most, if not all, of these steps into one expression. This makes code less error prone, easier to maintain, and may improve performance.

COMPONENT OBJECT INTERFACE

To integrate new data structures into the DATA step, the DECLARE statement was added along with an object dot syntax. This enables you to create an instance of a class, also called an object, then invoke methods on the object with a dot syntax, for example OBJ.METHOD(). The initial class provided is a hash table.

A hash table enables you to store data in an expandable, in-memory table that has fast lookup capabilities. It is similar to a DATA step array, where an index is used to store and retrieve data in the array. While an array uses numeric indices, a hash table can use character, numeric, or a combination of character and numeric indices. Unlike an array, the number of values stored in a hash table is not specified when creating the table. A hash table can grow to hold as many values as will fit into memory.

When adding data to a hash table, an index, often called a key, is used to find a position in the table to copy the data. When retrieving data from a hash table, a key is given and a fast, non-linear search occurs to determine if data with the key has been placed in the table. If so, the data stored with that key is returned.

A key is a vector of character, numeric, or both character and numeric values. Data are also a vector of character, numeric, or character and numeric values. For example, a character Social Security number (SSN) could be a key that maps to a person's name, address, and other information. When adding data, a SSN and a person's information would be added to the hash table. When retrieving data, a SSN is given and if data has been added to the table with that SSN, that data is returned.

This type of processing is not new with the DATA step, but is faster with a hash table. Using SET with KEY= is similar, where a key is given and a row is loaded into the PDV, but with a hash table, an index on the data set being searched is not needed and the lookup occurs in memory instead of on disk. MERGE with BY could also be used, but MERGE requires the input data sets to be sorted or indexed. Data loaded into a hash table need not be pre-sorted. Formats can be used as a lookup table method on large non-indexed data sets. A hash table is faster than using a format, uses less memory, and can store multiple data items per key. With large amounts of data, formats take up large amounts of disk space where hash tables do not.

The next sections will describe how hash tables are used from the DATA step.

USING A HASH TABLE FOR KEY LOOKUP

A common use for a hash table is to load it with key/data pairs, then query the hash table with keys from a data set. The following example shows how a hash table is loaded with a master data set and queried by a transaction data set. The master has variables `key` and `val`, and the transaction has variable `key`.

```
data merged;
  length key $ 13;
  length val 8;
  if _N_ = 1 then do;
    declare Hash ht(dataset: "master");
    ht.defineKey("key");
    ht.defineData("val");
    ht.defineDone();
  end;

  /* Load key and query hash table */
  set trans;
  if ht.find() = 0;
run;
```

In this example, a new hash table, `ht`, is declared, instantiated, and the constructor is instructed to load the hash table with the data set named `master`. A named parameter, `dataset`, is used to signify the data set to load the hash table with. Named parameters allow multiple parameters to be passed in any order and help document what a parameter represents. Similar to temporary arrays, objects are retained and dropped from any output data sets.

Before data can be loaded from a data set, the key and data variables for the hash table must be specified. The `defineKey` and `defineData` methods perform this action. We specified one key variable, `key`, and one data variable, `val`. Multiple key and data variables can be specified by passing more variable names to `defineKey` and `defineData`.

When the key and data variable definition is complete, the `defineDone` method is called. This causes the data set to be loaded into the hash table. Notice that the hash table is not sized. The hash table will grow in memory to accommodate the amount of data loaded.

During the query phase of the program, values from the transaction data set are loaded with the `SET` statement. The `find` method uses the current value of the key variables, performs a lookup in the hash table, and sets the values of the data variables if a match is found. The return value of the `find` method is zero for success and non-zero for failure.

KEY LOOKUP PERFORMANCE

To test performance, the key lookup example was recoded to perform the same search using a format, `SET` with `KEY=`, and `MERGE` with `BY`. The results are in the table below. For this example, the hash table takes 1.9 to 5.9 times less real time to perform the same lookup. With a format, time is spent creating the format and the format lookup takes longer than a hash table lookup. `SET` with `KEY=` incurs the cost of creating an index and the disk based lookup is slower than an in memory lookup. `MERGE` with `BY` performs well, but the input data sets must be sorted or indexed. A hash table is a good choice for lookups in unordered data that can fit into memory.

Real Time to Perform Lookup	
SET with KEY=	108.14s
Build Index	11.33s
Search	96.81s

Format and PUT()	89.03s	
Build Format		67.32s
Search		21.71s
MERGE with BY	35.78s	
Sort Data Sets		31.60s
Search		4.18s
Hash Table	18.35s	
Load and Search		18.35s

For each test, the total time is on the first line. Indented lines break down where time was spent for each operation. The times are real seconds reported by SAS when run on the SAS 9.1 for Windows. The best of three runs is presented. The test was run on a 1.6Ghz Pentium 4 machine with 1GB of RAM running Windows XP Professional. There are 5 million key/data pairs in the data set `master` and 5 million search keys in the data set `trans`. The key is a 13 character value and the value is an 8 byte numeric. Half of the search keys exist in the master data set.

ADDING DATA

Data for a hash table can be loaded from sources other than a data set. If no data set name is passed to the hash table constructor, an empty hash table is instantiated and the `add` method can be used to place data into the hash table. The following example adds data from an input file and performs a search after the data has been loaded.

```
data _null_;
  length first last title $ 16;
  length born died 8;
  if _N_ = 1 then do;
    declare Hash ht();
    ht.defineKey("first", "last");
    ht.defineData("born", "died", "title");
    ht.defineDone();
  end;

  infile datalines eof=search;
  input first last born died title & $16.;

  ht.add();
  /*
   ht.add() is the same as:
   ht.add(key:first, key:last,
         data:born, data:died,
         data:title);
  */
  return;

search:
  rc = ht.find(key: "John", key: "Keats");
  if rc = 0 then
    put "Found John Keats " title $QUOTE.;
  datalines;
  William Blake 1757 1827 Spring
  John Keats 1795 1821 To Autumn
  Mary Shelley 1797 1851 Frankenstein
  ;
```

Output:

```
Found John Keats "To Autumn"
```

If the `add` method is invoked with no parameters, it will copy the current values of the key and data variables into the hash table. If parameters are given, those values will be used in the hash table. The order of parameters must match the order that keys and data are defined. For instance, if the `born` and `died` parameters were swapped when passed to the `add` method, the birth year would be put in `died` and `born` would contain the year of death when data is retrieved.

This example also shows multiple values per key and multiple values per datum. Note that the data items are of mixed type.

HASH TABLE OUTPUT AND SORTING

After populating a hash table the data of the hash table can be persisted by using the output method to write the table to a data set. The data set can be utilized by other PROCs or loaded into a hash table by a future DATA step.

The rows output by the output method are not ordered unless the value "ascending" or "descending" is passed via the named parameter `ordered` when the hash table is created. When `ordered` is used, data items are ordered by the key. This is shown in the next example, which loads a hash table with patient ids and discharge dates. The hash table is output to the data set `sorted_ids` in sorted order by patient id.

```
data _null_;
  length patient_id $ 16 discharge 8;
  if _N_ = 1 then do;
    declare Hash ht(ordered:"yes");
    ht.defineKey("patient_id");
    ht.defineData("patient_id", "discharge");
    ht.defineDone();
  end;

  infile datalines eof=output;
  input patient_id discharge:DATE9.;
  ht.add();
  return;

output:
  ht.output(dataset: "sorted_ids");
datalines;
Smith-4123 15MAR2004
Hagen-2834 23APR2004
Smith-2437 15JAN2004
Flinn-2940 12FEB2004
;

data _null_;
  set sorted_ids;
  put patient_id discharge:DATE9.;
run;
```

Output:

```
Flinn-2940 12FEB2004
Hagen-2834 23APR2004
Smith-2437 15JAN2004
Smith-4123 15MAR2004
```

Note that only the data, not the keys, of a hash table are persisted by the output method. In order to persist the keys, the keys must be added to the data portion of the hash table. This is done in the prior example by using `patient_id` as both a key and data. If `patient_id` were only used as a key, it would not appear in the output data set.

HASH TABLE ITERATOR

In addition to the hash table component, there is a hash iterator component. A hash iterator enables sequential access to the data elements of a hash table. The hash table to iterate through is specified when a hash iterator is created. The example below loads patient ids and discharge dates into a hash table and uses a hash iterator to output each item in the hash table to the log.

```
data _null_;
  length patient_id $ 16 discharge 8;
  if _N_ = 1 then do;
    declare Hash ht(ordered:"yes");
    ht.defineKey("patient_id");
```

```
    ht.defineData("patient_id", "discharge");
    ht.defineDone();
  end;
```

```
  infile datalines eof=process;
  input patient_id discharge:DATE9.;
  ht.add();
  return;
```

```
  process:
    declare HIter iter('ht');
    rc = iter.first();
    do while (rc=0);
      put patient_id discharge:DATE9.;
      rc = iter.next();
    end;
  datalines;
  Smith-4123 15MAR2004
  Hagen-2834 23APR2004
  Smith-2437 15JAN2004
  Flinn-2940 12FEB2004
  ;
```

Output:

```
Flinn-2940 12FEB2004
Hagen-2834 23APR2004
Smith-2437 15JAN2004
Smith-4123 15MAR2004
```

This program is similar to the example that demonstrates the output method, except the data are output to the log instead of a data set. The `declare hiter` statement is used to create a hash iterator named `iter`. When the iterator is created, the hash table name to iterate over, 'ht', is passed to the iterator's constructor.

Once the iterator is created, there are four methods that can be used to retrieve data from a hash table: `FIRST`, `LAST`, `NEXT`, and `PREV`. In this example the `FIRST` method is used to get the data for the first item in the hash table. The `NEXT` method is used to retrieve the next item in the hash table. All of the iterator methods return 0 if an item is found and 1 if an item is not found. In this example, the `DO WHILE` loop exits when the `NEXT` method returns 1, which means there are no more elements in the hash table to return.

As in the `OUTPUT` method example, the named parameter `ordered` is set to "ascending" for the hash table iterated over. This causes the iterator to return data in sorted order based on the hash table key. If the `ordered` parameter wasn't set, the order the data is returned is not determinable.

HASH TABLE INTERNALS

A hash table is an array of buckets. When adding data or searching for a key, the key is passed to a hash function which returns a bucket number where the data is inserted or can be found. When multiple keys hash to the same bucket, the key/data pairs are stored in an AVL tree for that bucket.

If the number of key/data pairs placed into a hash table is larger than the number of buckets, performance is reduced since AVL trees will contain some of the items. Searching an AVL tree is slower than if all items were stored in hash table buckets.

In the opposite case, if the number of key/data pairs placed into the hash table is much smaller than the number of buckets, then many buckets will be empty. Unused buckets are a waste of memory. However, in many cases, the amount of wasted memory is small. For instance, the default number of buckets is 256 and the size of bucket is 8 bytes, so if half of the buckets are used, only 1024 bytes are wasted.

As a performance tuning parameter, the number of buckets used by the hash table can be changed. To change the number of buckets, the `hashexp` named parameter is used with the hash table constructor. The `hashexp` value denotes the number of buckets for the hash table in terms of a power of two. For example:

```
declare Hash ht(hashexp:6);
```

will instantiate a hash table with 2^{**6} , or 64, buckets. The maximum `hashexp` value is 16. Typically, millions of key/data pairs can be placed into a hash table with 2^{**8} or 2^{**9} buckets without a performance penalty. A hash table can grow until memory runs out, but tuning the number of buckets with large numbers of key/data pairs may help improve the performance of the add and find methods.

NEW_OPERATOR AND DELETE METHOD

In addition to creating an object with the `DECLARE` statement, the `NEW_OPERATOR` can also be used. The syntax for `NEW_OPERATOR` is to follow it with a class name and any constructor parameters. To free an existing object, the object's delete method is invoked. The delete method exists for every object and takes no parameters. The following example creates a hash table with `NEW_OPERATOR` and frees it with `delete`. A new hash table will be created and deleted for each implicit iteration of this `DATA` step. Note the use of the shorthand `DCL` for the `DECLARE` statement.

```
data _null_;
  length key1 data1 $ 8;
  length key2 data2 8;

  dcl Hash ht;
  ht = _new_ Hash ();
  ht.defineKey("key1", "key2");
  ht.defineData("data1", "data2");
  ht.defineDone();

  /* DATA step processing */

  ht.delete();
run;
```

DOCUMENTATION

The object dot syntax and hash table are experimental in the `DATA` step for SAS 9.0 and production in SAS 9.1. Documentation can be found in the `OnlineDoc` for SAS 9.1. Along with the methods already shown, the documentation explains the `remove` and `replace` methods and the `num_items` attribute for the hash table, which are not covered in this paper. The documentation also describes the `LAST` and `PREV` methods of the hash table iterator.

FUNCTIONS

The following sections highlight many new `DATA` step functions and `CALL` routines. Please refer to the "What's New" section of the SAS 9 Language Reference Dictionary for a complete list.

CALL SORTN, CALL SORTC

`CALL SORTN` and `CALL SORTC` sort the values of all the variables passed. `SORTN` sorts numeric variables and `SORTC` sorts character variables. With character variables, each must be the same size. When the routine returns, the variable values will be sorted in ascending order. `SORTN` and `SORTC` do not replace the `SORT` Procedure, but do give you a way to sort a small number of values within a `DATA` step. `SORTN` and `SORTC` are experimental in SAS 9.0 and 9.1.

```
data _null_;
  array rnd[5];
  do i = 1 to dim(rnd);
```

```
    rnd[i] = floor(ranuni(1)*100);
  end;
  call sortn(of rnd[*]);
  put "rnd[*] = (" rnd[*] +(-1) ")";
run;
```

Output:

```
rnd[*] = (18 25 39 92 97)
```

VVALUE, VVALUEX

`VVALUE` takes a variable and returns the value formatted as a character string. `VVALUEX` takes a string that contains a variable name and returns the value of the variable formatted as a character value. This is like using the `PUT` function without having to specify a format. The format used to format values is the default format for that type. For numeric values, the default format is typically `BEST12`. For character values, the default format is typically `$w.`, where `w` is the length of the value. The default format can be changed with the `FORMAT` statement.

```
data _null_;
  n = 123.456;
  a = vvalue(n);
  b = vvaluex('n');
  put "a = " a $12. / "b = " b $12.;
run;
```

Output:

```
a =      123.456
b =      123.456
```

CAT, CATS, CATT, CATX

The `CAT` functions concatenate and return the values passed. `CAT` concatenates values as if the concatenation operator, `||`, were used. `CATT` removes trailing blanks from each argument before concatenating (think `CAT` plus `TRIM`). `CATS` strips leading and trailing blanks before concatenating. `CATX` is like `CATS`, but places a delimiter between values being concatenated. Numeric values are formatted to character values using `BEST32`. For numbers with many digits and `BEST12`. For numbers with fewer digits. These functions reduce the need to use the `TRIM`, `LEFT`, and `PUT` functions with the concatenation operator.

```
data _null_;
  length a b c $ 8;
  a = 'some';
  b = ' text';
  n = 123.456;
  c = 'together';
  cop = trim(a) || trim(left(b)) ||
        trim(left(vvalue(n))) || c;
  cat = cat(a, b, n, c);
  catt = catt(a, b, n, c);
  cats = cats(a, b, n, c);
  catx = catx(' ', a, b, n, c);
  put +1 cop= / +1 cat= / catt= / cats= /
  catx=;
run;
```

Output:

```
cop=sometext123.456together
cat=some      text 123.456together
catt=some text123.456together
cats=sometext123.456together
catx=some, text, 123.456, together
```

SCANQ, COUNT, COUNTC

The `SCANQ` function parses delimited values that may contain a delimiter. `SCANQ` operates like the `SCAN` function, but ignores

delimiters that are inside of quoted text. The COUNT function takes two character parameters and counts the number of times the second parameter occurs in the first. The COUNTC function takes two parameters and counts the number of times each character in the second parameter occurs in the first. In this example, the COUNTC is used to obtain the number of commas in a character value.

```
data _null_;
  txt = "some,'comma,separated',text";
  put @2 'i' @5 'scan' @16 'scanq' / 32*'-';
  do i = 1 to countc(txt,',')+1;
    scan_word = scan(txt,i,',');
    scanq_word = scanq(txt,i,',');
    put @2 i @5 scan_word @16 scanq_word;
  end;
run;
```

Output:

```
  i  scan          scanq
-----
  1  some          some
  2  'comma      'comma,separated'
  3  separated' text
  4  text
```

In SAS 9.1, COUNT and COUNTC have an optional third argument for options. One of the options that can be specified is a case-independent search.

SUBSTRN, LENGTHN

The SUBSTRN function is like the SUBSTR function, but the length parameter can be zero. When the length parameter is zero, a zero length value is returned. SUBSTRN can be useful with regexp matches that may be zero length. The LENGTHN function is like the LENGTH function, but with a value that is all blanks, zero is returned. In this example, notice the NOTE output when SUBSTR is passed a zero length. SUBSTRN does not cause a NOTE like this to be output.

```
data _null_;
  length empty $ 8;

  substr = substr("some text", 1, 0);
  substrn = substrn("some text", 1, 0);
  put substr= / substrn=;

  length = length(empty);
  lengthn = lengthn(empty);
  put length= / lengthn=;
run;
```

Output:

```
NOTE: Invalid third argument to function
      SUBSTR at line XX column XX.
substr=some text
substrn=
length=1
lengthn=0
```

CALL SYMPUTX

CALL SYMPUTX operates like CALL SYMPUT, but it removes leading and trailing spaces from both the macro variable name and the data parameters. If a numeric value is passed as a second parameter, BEST32. is used to format numbers with many digits, otherwise BEST12. formats the number before placing it into the macro variable. SYMPUTX helps reduce DATA step notes about type conversion and remove unwanted spaces from macro variable values. In SAS 9.1, SYMPUTX adds an optional third argument to specify which scope (local or global) a new

macro variable should be placed.

```
data _null_;
  call symputx("CHARVAR1", " some text ");
  call symput ("CHARVAR2", " some text ");

  call symputx("NUMVAR1", 123.456);
  call symput ("NUMVAR2", 123.456);

  num = 12345678901234.123456;
  call symputx("NUMVAR3", num);
  call symput ("NUMVAR4", num);
run;
%put !&charvar1!      !&charvar2!;
%put !&numvar1!      !&numvar2!;
%put !&numvar3!      !&numvar4!;
```

Output:

```
NOTE: Numeric values have been converted to
character values at the places given by:
      (Line):(Column).
!some text!          ! some text !
!123.456!            ! 123.456!
!12345678901234.1!  !1.2345679E13!
```

MEDIAN, PCTL

The MEDIAN function returns the median of the non-missing values passed to the function. If all arguments are missing, the result is a missing value.

The PCTLN function takes a percentage and a list of values. PCTLN returns the percentile of the non-missing values corresponding to the percentage. N is a digit from 1 to 5 which specifies the definition of the percentile to be computed. If n is not specified, definition 5 is used.

```
data _null_;
  x=median(2,4,1,3);
  y=median(5,8,0,3,4);
  median=pctl(50,2,4,1,3);
  lower_quartile=pctl(25,2,4,1,3);
  percentile_def2=pctl2(25,2,4,1,3);

  put x= / y= / median=;
  put lower_quartile=;
  put percentile_def2=;
run;
```

Output:

```
x=2.5
y=4
median=2.5
lower_quartile=1.5
percentile_def2=1
```

The formulae used in the MEDIAN and PCTL functions are the same used in PROC UNIVARIATE.

FIND

The FIND function searches for one string within another and returns the position where a match is found. FIND is similar to INDEX, however FIND takes two more arguments, a position where to start searching and modifiers. If the start position is positive, a left to right search occurs. If the start position is negative, a right to left search occurs. The modifiers allow the caller to specify a case-independent search or for input arguments to be trimmed before the search occurs. The next example uses FIND to perform a case-independent, backward search for the text "CPU".

```
data _null_;
```

```

source = "Single cpu, Multi-CPU.";
location = find(source, "cpu", -999, "i");
put location=;
run;

```

Output:

```
location=19
```

The start position of -999 specifies a backward search from the end of the string and the modifier "i" specifies a case-independent search.

PROPCASE

In SAS 9.1, the PROPCASE function takes a string and returns the string with all characters converted to lowercase except for those that follow a list of delimiters. Those that follow a delimiter are converted to uppercase. The default delimiters are blank, forward slash, hyphen, open parenthesis, period, or tab. The optional second argument to PROPCASE allows users to specify an alternate set of delimiters.

```

data _null_;
input name $32.;
name = propcase(name);
put name=;
datalines;
MIKE LEARY
tonya bayer-macnie
GLoriA MaCDonaLd
;

```

Output:

```

name=Mike Leary
name=Tonya Bayer-Macnie
name=Gloria Macdonald

```

Notice how "Macdonald" should be cased as "MacDonald". PROPCASE cannot determine the proper case in this situation. The DQCASE function, part of the SAS Data Quality Server product can be licensed to handle names of this type.

SYMEXIST, CALL SYMDEL

The SYMEXIST function takes the name of a macro variable and returns 1 if a macro variable with this name exists and 0 if it does not exist. CALL SYMDEL takes a character argument that is the name of a macro variable. If the macro variable exists, it is deleted. The following example tests to see if macro variable MVAR1 exists, then it outputs the value of MVAR1 and deletes the variable.

```

%let mvar1 = Inventory;
data _null_;
if symexist('mvar1') then do;
str = symget('mvar1');
put str=;
call symdel('mvar1');
end;
run;
%put &mvar1;

```

Output:

```

str=Inventory
WARNING: Apparent symbolic reference MVAR1
not resolved.

```

CALL ALLPERM, CALL RANPERM, CALL RANPERK

These CALL routines compute permutations of their input arguments. ALLPERM computes all permutations of its input arguments. RANPERM randomly permutes its input values. RANPERK randomly permutes its input arguments and returns k out of n of them.

```

data _null_;
array vals[3] $ 3 ('PBS' 'NPR' 'PRI');

do i = 1 to fact(dim(vals));
call allperm(i, of vals[*]);
put i vals[*];
end;

put;
seed = 44;
do i = 1 to 4;
call ranperm(seed, of vals[*]);
put i vals[*];
end;
run;

```

Output:

```

1 PBS NPR PRI
2 PBS PRI NPR
3 PRI PBS NPR
4 PRI NPR PBS
5 NPR PRI PBS

1 PRI NPR PBS
2 PBS PRI NPR
3 PBS NPR PRI
4 NPR PBS PRI

```

An example of RANPERK can be found in the OnlineDoc. The FACT function computes a factorial and is used in the example to compute the number of permutations that ALLPERM computes.

PERFORMANCE ENHANCEMENTS

FASTER LENGTH AND TRIM FUNCTIONS

The LENGTH and TRIM functions have been optimized on all hosts to perform a more efficient end of value search. Instead of searching backward one character at a time to find the first non-blank character, 64-bit integers are used to search backward 8 bytes at a time. Using 64-bit integer comparisons reduces the number of instructions required to find the last non-blank character.

The improvement is best with long values that have many blanks. Improvements of 4x in execution time have been observed with the LENGTH and TRIM functions.

NO SPILL FILE VIEWS

When a DATA step view is opened in random, two pass, or BY group rewind mode, the view opens a spill file that will contain all of the observations it has output. The spill file is used to retrieve prior observations that may be requested. With views that return large amounts of data, large amounts of disk space are needed. A problem occurs if there is no additional disk space.

When a view is opened for two pass, a spill file is not necessary. Instead, the view can be restarted for each pass through the data. When a view is opened for BY group rewind, as is done by procedures that iterate over values within a BY group, only the current BY group must be spilled, reducing the amount of disk space required for a spill file.

An experimental data set option `SPILL=[NO|YES]` has been added to DATA step views to tell a view to not produce spill files when opened for two pass mode and only spill the current BY group when opened in BY group rewind mode. When opened in random mode, a spill file is still created. The default is `SPILL=YES`, the same spill behavior as prior versions of SAS.

When `SPILL=NO` and the view is opened with two pass, the view

is restarted when a prior observation is requested. There are several implications with restarting a view. Any output to an external file or data set by the view will be repeated. Also, non-deterministic views may produce different results for each pass through the data. Non-determinism can affect a view if a random or time function is used within the view.

New INFO messages report when a spill file is created or deleted and when a view is restarted. INFO messages are enabled by using the global option MSGLEVEL=I. The following example shows how the SPILL= data set option is used to print the DATA step view dsv without producing a spill file.

```
proc print data=dsv(spill=no) width=uniform;
```

DATA STEP LANGUAGE EXTENSIONS

“IN” SEARCH WITH INTEGER RANGES AND ARRAYS

Integer ranges and arrays can be searched with the IN operator. Integer ranges can also be used to initialize an array with the ARRAY and RETAIN statements. An integer range, the integers from M to N inclusive, is specified with M:N. The following example shows how to use integer ranges.

```
data _null_;
  array arrA[10] (1:10);
  array arrB[10];
  retain arrB (2*1:5);

  put 'arrA=' arrA[*] / 'arrB=' arrB[*];

  x = 5;
  if x in (1:10000) then
    put 'X in range 1-10000';

  if x in (1 2 5:10) then
    put 'X in 1, 2, 5-10';

  if x in arrA then
    put 'X in arrA';
run;
```

Output:

```
arrA=1 2 3 4 5 6 7 8 9 10
arrB=1 2 3 4 5 1 2 3 4 5
X in range 1-10000
X in 1, 2, 5-10
X in arrA
```

When an integer range is searched with the IN operator, an integer check and a comparison against the lower and upper bounds occurs. This check is faster than checking against all values within the range.

PUTLOG

The PUTLOG statement is similar to the PUT statement, but all of its output is sent to the SAS log instead of the current file destination. The PUTLOG statement is handy within macros, where you don't know where the macro will be used and if a non-log external file destination is current. If the first characters output are “NOTE:”, “WARNING:”, or “ERROR:” the output will be colored appropriately in the log.

```
data _null_;
  file tmp;
  input lastName $;
  put lastName;
  if lastName = 'J' then
    putlog 'NOTE: J last name, ' lastName;
datalines;
Harris
```

```
Jones
Barber
Johnson
;
```

Output:

```
NOTE: J last name, Jones
NOTE: J last name, Johnson
NOTE: 4 records were written to the file TMP.
      The minimum record length was 5.
      The maximum record length was 7.
```

SAS MACRO LANGUAGE OPTIONS

Several new SAS Macro system options are not DATA step enhancements, but they deserve some mention since DATA step programmers may find them helpful.

OPTION MCOMPILENOTE = [NONE | ALL | NOAUTOCALL]

When option MCOMPILENOTE is enabled, SAS will output a note to the log when a macro is successfully compiled. Valid values for this option are NONE, ALL, and NOAUTOCALL. NONE is the default, ALL displays a note when any macro is compiled, and NOAUTOCALL outputs a note when non-autocall macros are compiled.

```
options mcompilenote=all;
%macro mymacro;
  data _null_; x=1; put x=; run;
%mend;
NOTE: The macro MYMACRO completed compilation
      without errors.
```

OPTION MPRINTNEST | NOMPRINTNEST

When using option MPRINTNEST with option MPRINT, macro nesting information is displayed with MPRINT log output. In the example below, the macro MAC1 is nested within MAC2. When MAC1 is expanded within MAC2, MPRINT indicates this by using the form MPRINT (MAC2.MAC1).

```
options mprint mprintnest;
%macro mac1;
  put 'mac1';
%mend;

%macro mac2;
  %mac1;
  put 'mac2';
%mend;

data _null_;
  %mac2;
run;

MPRINT (MAC2.MAC1):   put 'mac1';
MPRINT (MAC2):       ;
MPRINT (MAC2):       put 'mac2';
```

OPTION MAUTOLOCDISPLAY | NOMAUTOLOCDISPLAY

When option MAUTOLOCDISPLAY is used, the location of the source for an autocall macro is output as a note when the autocall macro is used. This can be helpful in debugging path problems with autocall macros.

CONCLUSION

The new features and enhancements were added to ease programming, improve execution time, and enable new types of programs to be written. The addition of Perl regex and a hash table component will speed and simplify many DATA steps. Most of the new functions simplify common DATA step operations. The new SPILL= data set options for views helps reduce or eliminate the amount of disk space a view requires. All of the new features

and enhancements came from user suggestions. We invite your feature requests. Please call Technical Support or email your name, phone number, and feature request along with a motivating example to suggest@sas.com.

REFERENCES

[1] Dorfman, P. 2000. Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing. SESUG 2000. P-105.

[2] Friedl, J. 2002. Mastering Regular Expressions, Second Edition. O'Reilly & Associates, Sebastopol, CA.

[3] Wall, L., Christiansen, T., and Orwant, J. 2000. Programming Perl, 3rd Edition. O'Reilly & Associates, Sebastopol, CA.

ACKNOWLEDGMENTS

The author would like to express his appreciation to the following people for developing the features discussed in this paper, and/or for their assistance with this paper: Bill Heffner, Al Kulik, Robert Ray, Warren Sarle, Bonnie Horne, Ed Vlazny, Diana Fox, Kevin DeBruhl, Rick Langston, Janice Bloom, Michelle Schlude, Kevin Hobbs, Charley Mullin, Ginny Piechota, Linda Reznikiewicz, and Chris Olinger.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Jason Secosky
SAS
SAS Campus Drive
Cary, NC 27513
(919) 677-8000
jason.secosky@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.