

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

Jason Secosky, SAS, Cary, NC.

INTRODUCTION

This paper presents DATA step language enhancements in SAS Version 9. These enhancements include: Perl regular expressions for fast search and replace in text, hash tables for searching a 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 a popular and powerful search, replace, and extraction tool 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 are error prone and make searching dynamic text difficult, if not impossible. Regex can combine several operations into a single function call. The following sections discuss common cases for using regex in data validation, replacing text, and extracting text.

INTRODUCTION

A regex is a string of characters and special characters called metacharacters. When performing a regex match, a 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. Since blanks in a regex will match blanks in a string and to make the regex bounds clear, regex are always surrounded by non-alphanumeric delimiters. The most common delimiter is forward slash, `/`.

The regex metacharacters perform special actions when matching, 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 or on perldoc.com.

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 parse a regex 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 matched the value, or zero if no match was found.

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

```
length first last phone $ 16;
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.

REPLACING TEXT

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

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

```
data _null_;
  if _N_ = 1 then do;
    retain re;
    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.

EXTRACTING TEXT

Parentheses are regexp metacharacters that enable extracting part of a regexp match. Placing parentheses around part of a regexp will enable you to retrieve the position and length of where that part of the regexp matched. For example, the regexp `/(\d\d\d)-(\d\d\d)/` will match the text 123-456. Submatch #1 is 123 and submatch #2 is 456. Calling PRXPOSN with the number of the submatch will retrieve the position and length of the 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_;
  if _N_ = 1 then do;
    retain re;
    re = prxparse("/\b([2-9]\d\d)\b ?" ||
      "[2-9]\d\d-\d\d\d\d/");
  end;

  length first last phone $ 16;
  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
```

PRX FUNCTIONS

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

- CALL PRXSUBSTR returns the position and length of a match without using PRXPOSN.
- 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 parsed regexp.
- CALL PRXDEBUG toggles output of Perl debug information to the SAS log.

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, easily 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.

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, clearer, and may improve performance.

OBJECT DOT SYNTAX

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 first class provided is a hash table.

A hash table uses a fast, non-linear search to index unsorted data with a key. Hash tables perform their search in memory. A key is a vector of numeric, character, or numeric and character values. Each key maps to a data vector of numeric, character, or numeric and character values. For example, a character Social Security number could be a key that maps to a person's name, address, and other information.

This type of processing is not new with the DATA step. Using SET with KEY= is similar, 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. Formats can be used as a lookup table method on large non-indexed data sets. A hash table is faster than using a format, takes 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 takes 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 2.5 to 14 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.

Format and PUT()	22.84s	
Build Format		16.03s
Search		6.81s
SET with KEY=	47.85s	
Build Index		3.06s
Search		44.79s
MERGE with BY	9.18s	
Sort Data Sets		7.58s
Search		1.60s
Hash Table	3.43s	
Load and Search		3.43s

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 Early Adopter release of SAS Version 9 for Windows. The best of three runs is presented. The test was run on a 1.5Ghz Pentium 4 machine with 512MB of RAM running Windows XP Professional. There are 1 million key/data pairs in the data set `master` and 1 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 array. 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, the birth year would be put in `died` and `born` would contain the year of death.

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

HASH TABLE DETAILS

With a hash table, the key is passed to a hash function which returns a bucket number where the data 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, the AVL tree for each bucket will probably contain more than one item. In this case, the lookup time for the hash table will be longer than if there were more buckets, because an AVL tree must be searched.

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 could 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

To create a new object, the NEW operator is used. The syntax for NEW is to follow it with a class name and any constructor parameters. To free an existing object, the delete method is invoked. The delete method exists for every object and takes no parameters. The following example creates a hash table with NEW and frees it with delete. A new hash table will be created and deleted for each implicit iteration of the 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 Version 9. This means that extensive testing has not occurred and documentation is not included with SAS. Documentation can be found online at <http://www.sas.com/rnd/base> in the DATA step section. The documentation includes the replace method and information about instantiating an iterator for a hash table.

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 SORTQ

CALL SORTQ sorts the values of all the variables passed. All of the variables must be of the same type. With character variables, each must be the same size. When the routine returns, the variable values will be sorted in ascending order. SORTQ uses Quicksort to sort the values. SORTQ does not replace the SORT Procedure, but does give you a way to sort a small number of values within a DATA step. SORTQ is experimental in SAS Version 9.

```
data _null_;
  array rnd[5];
  do i = 1 to dim(rnd);
    rnd[i] = floor(ranuni(1)*100);
  end;
  call sortq(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. 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,separted',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,separted'
3 separted' text
4 text

```

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.

```

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.

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.
```

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 regexp and a hash table class 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

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

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

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, 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.