Using Recursion to Trace Lineages in the SAS® ODS Styles.Default Template

Perry Watts, Independent Consultant, Elkins Park, PA

ABSTRACT

As a first step towards learning how to generate customized tables in ODS, it is recommended that the Styles.Default template be written out to the SAS LOG. What greets you in 9.1.3 SAS is a 700-line listing of default values for attributes that are building blocks for 150+ style elements. That the template listing is overwhelming is an understatement. You still don't have a clue about how style templates generally work in ODS, but you realize that a comprehensive guide can only be found in the output you are looking at.

To translate the ODS Styles.Default template into something useful, inheritance has to be fully expressed. Whole lineages of style elements can then be traced and presented alongside their associated attributes. With the newly formatted guide, you can easily learn what is available when ODS-generated tables require a new appearance.

Inheritance falls out when recursion is applied to the Styles.Default template. In this paper, recursion is introduced by showing brief code examples written in macro and PROC SQL. Next, the DATA step that processes Styles.Default recursively is reviewed in detail. Final output is produced in ODS with an HTML destination. Usage of the new guide is deferred to a follow-up paper.

PROBLEM DEFINITION

There is no way to organize the Styles.Default template by lineage. Instead, members of any given lineage are widely scattered throughout the template. For example in Figure 1, lines 164 to 479 from the template must be scanned to locate the five ancestors for the HEADEREMPHASISFIXED style element. The only structural rule that can be counted upon is that ancestors precede their descendents in the Styles.Default template.

Figure 1. A partial listing of the ODS Styles.Default template is presented in linear order. Lines 1 to 163 define styles that have no ancestors. Examples include FONTS, GRAPHFONTS and COLOR LIST. CONTAINER also has no ancestors (the FROM clause is missing) but all styles of interest in this paper inherit directly or indirectly from CONTAINER. Inheritance below is traced by arrows. The highlighted lineage is complete, since HEADEREMPHASISFIXED has no descendents.

```
164    style Container
165       "Abstract. Controls all container oriented elements." /
166       font = Fonts('DocFont')
167       foreground = colors('docfg')
168       background = colors('docbg');
...
414    style Cell from Container
415       "Abstract. Controls general cells.";
...
450    style HeadersAndFooters from Cell
451       "Abstract. Controls table headers and footers." /
452       font = fonts('HeadingFont')
453       foreground = colors('headerfg')
454       background = colors('headerbg');
...
465    style Header from HeadersAndFooters
466       "Controls the headers of a table.";
...
472    style HeaderEmphasis from Header
473       "Controls emphasized table header cells." /
474       foreground = colors('headerfgemph')
475       background = colors('headerbgemph')
476       font = fonts('EmphasisFont');
477    style HeaderEmphasisFixed from HeaderEmphasis
478       "Controls emphasized table header cells. Fixed font." /
479       font = fonts('FixedEmphasisFont');
```

The reason why lineage listing is so difficult to implement is that each style element definition appears only once in the Styles.Default template. What is needed is a judicious use of redundancy. In Figure 2, CONTAINER appears four times as it should; once for each lineage that uses it. Recursive processing can be adjusted so that redundancy is inserted into the display where needed.

Figure 2. Four of the 60 Container lineages are derived recursively from the Styles.Default Template.

```
Lineage     Line#             Style Element
-------     -----    ------------------------------------
16            164      style Container
              414      style Cell from Container
              450      style HeadersAndFooters from Cell
              465      style Header from HeadersAndFooters
              472      style HeaderEmphasis from Header
              477      style HeaderEmphasisFixed from HeaderEmphasis

26            164      style Container
              214      style Date from Container
              227      style ContentsDate from Date

32            164      style Container
              169      style Index from Container
              250      style IndexProcName from Index
              260      style ContentProcName from IndexProcName
              262      style ContentProcLabel from ContentProcName

60            164      style Container
              305      style TitlesAndFooters from Container
              319      style SystemTitle from TitlesAndFooters
```

When style elements are listed in lineage order, the developer can quickly identify default values assigned to affiliated attributes. The 60 Container lineages span lines 164 to 544 in the Styles.Default template. (Graphics lineages do not inherit from Container. Instead they inherit from GraphComponent which has no ancestor).

RECURSION DEFINED

From Tannenbaum[4, p. 96-98] and Wikipedia[7] the definition for recursion contains two major components:

1) Recursion solves problems by finding solutions to smaller instances of the same problem. This approach to problem solving is iterative, because it repeats the same process until a predefined condition is met.
2) The predefined condition is called the stopping point or "base case" where no reference is made to a smaller instance of the same problem. Otherwise, processing would become infinite.

The trail of recursive processing can be graphed as an upside-down tree where the root or top node represents the base case, and the leaves represent starting points for recursive processing.

Recursion is illustrated below by looking first at two classic examples recreated in SAS. Art Carpenter shows how to write n! (n-factorial) recursively as a macro [1, p. 394], and a highly abbreviated report of the chain of command in President Obama's cabinet is created with a self-join in PROC SQL. The self-join is a limited version of a fully recursive join available in Oracle®.

EXAMPLE #1: THE FACTORIAL FUNCTION

Art Carpenter's algorithm for n-factorial is reproduced in the source code that follows [1, p. 394]:

```
%macro factorial(number);
  %if &number gt 1 %then %eval(&number * %factorial(%eval(&number-1)));
  %else 1;
%mend factorial;

options nosymbolgen mlogic mlogicnest;
%put The factorial of 5 is: %factorial(5);
```

This example shows how recursion works. By creating a function that calls itself, n-factorial is solved by finding solutions to smaller instances of the same problem. 5-factorial in this call = 5*4!(4!= 4*3!)(3!=3*2!)(2!=2*1!)(1=1). Each time FACTORIAL is invoked, NUMBER is successively decremented by 1. When NUMBER reaches 1 the stopping condition kicks in **(%else 1)** and the function calls execute as 1*2*3*4*5=120.

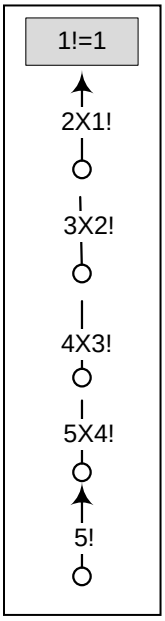
Carpenter's solution works in SAS, because both FACTORIAL and EVAL are macro *functions* that have return values. Unlike C, you cannot assign the output of a function call to a variable in SAS. In other words, **%let**

FACT=%eval(&number * %factorial(%eval(&number-1)))doesn't work. Likewise, a variable assignment is not needed in the PUT statement containing the embedded macro call. **%factorial(5)** simply resolves to 120.

The MLOGICNEST option in the code for n-factorial along with a tree containing a single branch in Figure 3 shows how SAS works recursively in macro.

Figure 3. Each time FACTORIAL is called, execution BEGINS anew. Eventually five FACTORIAL calls are strung together; one per macro call. When NUMBER is reduced to 1, the stopping condition is activated and execution ENDS for the last entered BEGIN (This is an example of LIFO processing). Executions continue to END until all the FACTORIAL calls are resolved.

```
48   %put The factorial of 5 is: %factorial(5);
MLOGIC(FACTORIAL):
  Beginning execution.
MLOGIC(FACTORIAL):
  Parameter NUMBER has value 5
MLOGIC(FACTORIAL):
  %IF condition &number gt 1 is TRUE
MLOGIC(FACTORIAL.FACTORIAL):
  Beginning execution.
MLOGIC(FACTORIAL.FACTORIAL):
  Parameter NUMBER has value 4 %EVAL(5 * FACTORIAL(4))
MLOGIC(FACTORIAL.FACTORIAL):
  %IF condition &number gt 1 is TRUE
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL):
  Beginning execution. %EVAL(5 * %EVAL(4 * FACTORIAL(3)))
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL):
  Parameter NUMBER has value 3
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL):
  %IF condition &number gt 1 is TRUE
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Beginning execution. %EVAL(5 * %EVAL(4 * EVAL(3 * FACTORIAL(2))))
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Parameter NUMBER has value 2
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  %IF condition &number gt 1 is TRUE
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Beginning execution. %EVAL(5 * %EVAL(4 * EVAL(3 * EVAL(2 * FACTORIAL(1)))))
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Parameter NUMBER has value 1
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
%IF condition &number gt 1 is FALSE
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Ending execution. %EVAL(5 * %EVAL(4 * EVAL(3 * EVAL(2 * 1))))
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL.FACTORIAL):
  Ending execution. %EVAL(5 * %EVAL(4 * EVAL(3 * 2)))
MLOGIC(FACTORIAL.FACTORIAL.FACTORIAL):
  Ending execution.%EVAL(5 * %EVAL(4 * 6))
MLOGIC(FACTORIAL.FACTORIAL):
  Ending execution.%EVAL(5 * 24)
MLOGIC(FACTORIAL):
  Ending execution. 120
```



EXAMPLE #2: USING RECURSION TO DISPLAY AN ORGANIZATIONAL HIERARCHY

A self or reflexive join in PROC SQL is used to display the chain of command in an organization. In this example, President Obama's cabinet has been reduced to two secretaries, and each of the two secretaries has just two subordinates; a perfect binary tree! First, the input data set:

```
data govtHierarchy;
  Length EName Title $15;
  infile cards missover;
  input ENum EName Title MNum ; /* Prefixes 'E'= Employee and 'M' = Manager */
  cards;
  2 Clinton    SecyState        1
  3 SteinBerg  DepSecyState     2
  6 Lynn       DepSecyDefense   5
  1 Obama      President
  4 Mills      ChiefOfStaff     2
  5 Gates      SecyDefense      1
  7 Mullen     JntChiefsStaff   5
run;
```

Unlike *n-factorial*, the input data do not have to be entered in strict order. For example, President Obama's entry appears in the middle of the data set – not in the beginning. However, for the SQL join to work in SAS, employee numbers have to be assigned in "left-branch" order so that output is organized by lineage affiliation. By way of illustration, employee numbers are affixed to the tree displayed in Figure 4. For a discussion of the left-branch tree traversal algorithm, see [5, p. 780-782].

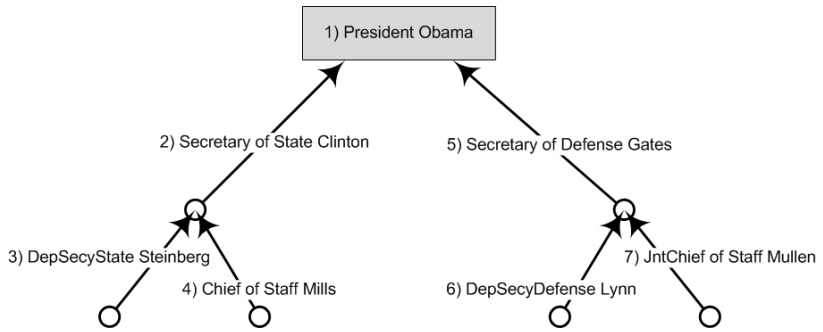
Recursion in this example takes advantage of the fact that managers are also employees. So looking at a smaller instance of the same problem, let's find out who is Steinberg's boss. That would be Secretary of State Clinton. Then who is Clinton's boss? That would be President Obama. Since President Obama represents the "base case", execution stops for this lineage. When all the "starting points" (Steinberg, Mills, Lynn, and Mullen) are processed, the program terminates.

In PROC SQL below, the input data set, **govtHierarchy**, is being joined with itself. The ORDER clause guarantees that the output in Figure 4 is printed "depth-first" or by lineage.

```
proc sql;
  select LowerH.Enum as EmpNum, LowerH.EName as EmpName,
         LowerH.Title as EmpTitle,
         UpperH.Enum as MgrNum, UpperH.EName as MgrName,
         UpperH.Title as MgrTitle
  from govtHierarchy as LowerH, govtHierarchy as UpperH
  where LowerH.MNum eq UpperH.ENum
  order by LowerH.ENum;
quit;
```

Figure 4. The chain of command in the Obama administration maps to a binary tree with four lineages and three levels. President Obama as the "base case" occupies the root node. Leaf nodes with Steinberg to Mullen would be identified as "starting points" for recursion.

EmpNum	EmpName	EmpTitle	MgrNum	MgrName	MgrTitle
2	Clinton	SecyState	1	Obama	President
3	SteinBerg	DepSecyState	2	Clinton	SecyState
4	Mills	ChiefOfStaff	2	Clinton	SecyState
5	Gates	SecyDefense	1	Obama	President
6	Lynn	DepSecyDefense	5	Gates	SecyDefense
7	Mullen	JntChiefsStaff	5	Gates	SecyDefense



In contrast to SAS, ORACLE has implemented recursion in a way that does not require an assignment of a pre-ordered identification number or a self-join [3, p. 275-286]. Instead, the FROM keyword associated with a single table is coupled with ORACLE commands START, CONNECT BY, and PRIOR to produce the following:

Manager	Subordinates
Obama	Clinton
Clinton	Steinberg
Clinton	Mills
Obama	Gates
Gates	Lynn
Gates	Mullen

An Oracle-type solution where ordering is determined exclusively by relationships among data elements is required for creating the ODS lineage tracer. In this instance, PROC SQL is replaced with a couple of SET statements having POINT options.

TRACING ODS STYLE ELEMENTS BY LINEAGE

Of interest to the user of *n-factorial* is the starting point (5!) or more correctly, its return value of 120. Values for intermediate points are only displayed to show how recursion works behind the scenes. For the *chain of command* problem, however, the entire tree is the object of interest. What the tree provides is a comprehensive view of employee relationships: both by lineage (top down, hierarchical) and by level (across, peer).

For the ODS lineage-tracer, each lineage becomes the object of interest. In "lineage-trace mode", output from Figure 4 would be reconfigured as:

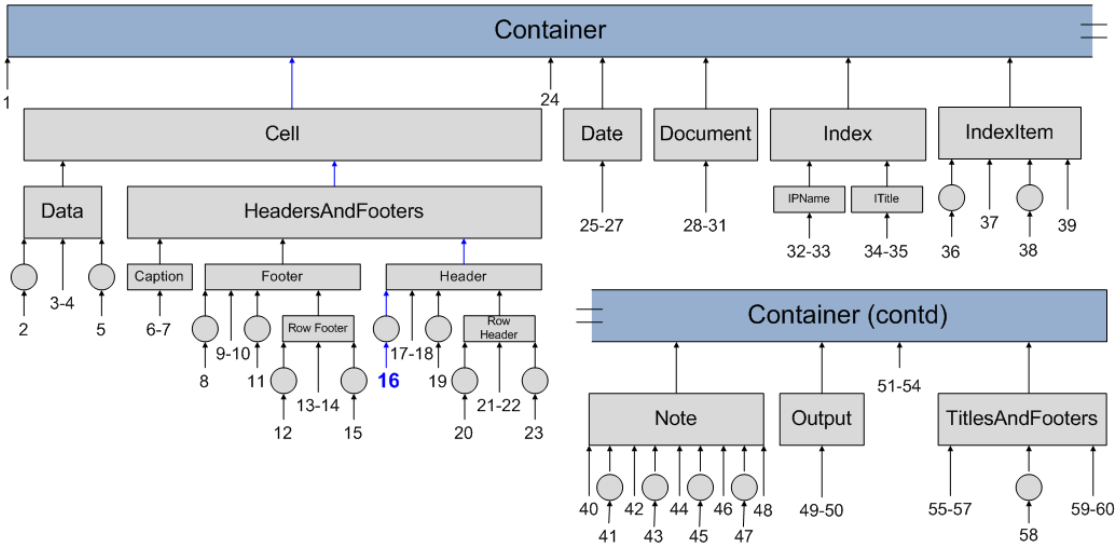
Lineage	Official #1	Official #2	Official #3
1	President Obama	Secretary of State Clinton	Deputy Secretary of State Steinberg
2	President Obama	Secretary of State Clinton	Chief of Staff Mills
3	President Obama	Secretary of Defense Gates	Deputy Secretary of Defense Lynn
4	President Obama	Secretary of Defense Gates	Joint Chiefs of Staff Mullen

The single lineage from Figure 1 becomes Lineage #16 in the lineage tracer:

Lineage	Node #1	Node #2	Node #3	Node #4	Node #5	Node #6
16	Container	Cell	HeadersAndFooters	Header	HeaderEmphasis	`HeaderEmphasisFixed`

Recursion in this example takes advantage of the fact that both *style* and *from* clauses in the Styles.Default template reference style elements. With linked style elements, it becomes possible to define a lineage recursively by looking at smaller instances of the same problem. Working backwards, to follow the direction of the arrows in figure 1, HEADEREMPHASISFIXED inherits attributes **from** HEADEREMPHASIS. Next, HEADEREMPHASIS inherits **from** HEADER, and so on up to CONTAINER which is the base case. When the *from* clause equals CONTAINER, a single lineage is defined, and when *style* clause equals CONTAINER, processing terminates. The 60 lineages derived from CONTAINER in the ODS Styles.Default template map to the tree displayed in Figure 5.

Figure 5. The CONTAINER tree is wide and shallow. Leaf nodes in the tree reference unique style elements. They are represented by lineage number rather than by the *style* name to save space. Circles also serve as place holders for embedded style elements. Lineage #16 shown earlier has been highlighted in blue for easy reference.



CREATING THE LINEAGE-TRACER

To translate the ODS Styles.Default template into a list of stand-alone CONTAINER lineages, five steps must be followed in strict order:

1) Read in the raw file, **stylesDefault.txt** and save CONTAINER style elements to two SAS data sets: **containerStyles** and **containerAttributes**. **containerStyles** has three variables: *styleNum*, *styleName* and *fromName*. *styleName* and *fromName* are parsed from the original style element definition in **stylesDefault.txt**. For example, *styleName* = **`Pages`** and *fromName* = **`Document`** when the style element definition = **`Pages from Document.`** **containerAttributes** also has three variables: *styleName*, *attributeNum* and *attribute*. Separating style elements from attributes simplifies the recursive processing that is only applied to **containerStyles**. **containerAttributes** becomes the data source for detail entries displayed in Figure 8 below.

2) Sort **containerStyles** in descending order so that the "starting points" or descendents precede their ancestors. The only motivation for the sort is to simplify data processing.

3) Create the **lineages** data set by applying recursion to **containerStyles**. Variables saved to the **lineages** data set include *numNodes* and *node1-node7*. *numNodes* contains the number of style elements in a given lineage. The number ranges from 2 to 7. From the CONTAINER tree in Figure 5, lineage #1 contains two elements, and lineage #20 contains seven. Values in *node1-node7* reference style elements that define a single lineage. The **lineages** data set contains sixty observations: one for each lineage.

4) Post-process the **lineages** data set to reverse the effects of the descending sort in step #2. *node7* becomes *style1* in the new data set. Then *node6* becomes *style2*, node5 becomes *style3* and so on. The output data set, **reverseLineages**, contains the re-ordered variables, *style1-style7.*

5) Sort **reverseLineages** by *style2 … style7*. (*style1* is always set to "Container"). Add a lineage number, *linNum*, to the data. The variables in the final data set, **containerLineages**, include *linNum* and *style1-style7*. See Figure 6 for an illustration of the data transformation that takes place between step #3 and step #5.

STEP#3: CREATE THE LINEAGES DATA SET WITH RECURSION

To explain how recursion is used in step #3, a small input data set with just 8 observations has been derived from the larger **containerStyles** data set. From Step #2 above, the input data are sorted in descending order so that processing can proceed in a simplified top-down fashion. Descendents appear before their ancestors in **rContainer** (check *styleNum*) with the result that the CONTAINER style element now appears at the end of the data set. Even with such a small data set, it is difficult to see that three complete lineages will be fully defined with an application of recursion.

Data Set: rContainer

styleNum	styleName	fromName
83	HeaderEmphasisFixed	HeaderEmphasis
82	HeaderEmphasis	Header
80	HeaderFixed	Header
79	Header	HeadersAndFooters
75	HeadersAndFooters	Cell
64	Cell	Container
39	BylineContainer	Container
10	Container	

Key to understanding the data step for step#3 is the double looping in the code that supports two SET statements and their associated stopping conditions. The DO-Loops, SET, and stopping conditions are highlighted in yellow below.

In the outer loop, the input data are processed in sequential order from the first observation to the last where the stopping condition, ***styleName* EQ "Container"**, is satisfied. Each time a record is read in from the outer-loop SET statement, an inner loop SET statement is conditionally executed to identify a complete lineage. The inner loop starts searching for lineage members at record **i+1** where **i** is reset each time the outer loop executes. Each lineage is fully defined when ***fromName* EQ "Container".**

Processing is conditional for lineage definition in the LINEAGES data step. To prevent redundant partial lineages from being generated, the value for *oldFromName* must not be the same as the value for *node[1]*, derived from the current value for *styleName*. In the code listing below, *oldFromName* entries are highlighted in blue. Interim output in Figure 6 shows how *oldFromName* does its job.

```
/* STEP #3 DATA PROCESSING */
data lineages (keep= numNodes node1-node7);
  array node {7} $30 node1-node7;  /* FOR UP TO 7 STYLE ELEMENTS PER LINEAGE */
  length oldFromName $30;
   retain node1-node7 k;
   start=1;
   do i=start to totobs;
     /* OUTER LOOP: READ IN LEAF NODES FIRST */
     set rContainer point=i nobs=totobs;
     /* STOPPING POINT FOR EXECUTION -- WHERE STYLENAME="CONTAINER" */
     if styleName EQ 'Container' then stop;
     else do;
      /* HOUSEKEEPING OCCURS AFTER THE CREATION OF A COMPLETE LINEAGE */
       if i GT start then do;
         oldFromName=node[2];
         do kk=1 to 7;
           node[kk]=' ';
         end;
       end;
       node[1]=stylename;
```

```
      node[2]=fromName;
     /* FOR LINEAGES WITH ONLY TWO STYLE ELEMENTS. */
      if fromName eq 'Container' AND stylename NE oldFromName then do;
        NumNodes=2; output;
      end;
      else do;
       /* EXECUTE THE INNER LOOP FOR LINEAGES HAVING MORE THAN TWO STYLE ELEMENTS.
          DATA ARE POLLED IN THE INNER LOOP FROM [CURRENT_REC + 1] TO TOTOBS. */
        iplus1=i+1;
        k=2;
        if node[1] NE oldFromName OR i EQ start then do j=iplus1 to totobs;
         /* CREATE A COMPLETE LINEAGE */
          set rContainer point=j nobs=totobs;
          if styleName EQ node[k] then do;
            k=k+1;
            node[k]=fromName;
            if node[k] eq 'Container' then do;
              NumNodes=k; output;
              leave;   /* STOPPING POINT FOR LINEAGE DEFINITION -- WHERE FROMNAME="CONTAINER" */
            end; *if CONTAINER;
          end; *if stylename;
        end; *do j (inner loop);
      end; *else do recursive search for one entry;
    end; *else do go to next entry;
  end; *do i (outer loop);
run;
```

Figure 6. Interim output from step #3 and final output from step#5 are displayed. Only the blue complete lineages are written out to the **lineages** data set in step #3. Overlooked are partial lineages where *oldFromName* **is** equal to node1. Also, as expected from the outer loop SET statement, *node1* and *node2* are identical in value to *styleName* and *fromName* in **rContainer**. In step#5 *node1* - *node7* (descendent →ancestor) is replaced with *style1* - *style7* (ancestor → descendent). The final data are sorted alphabetically by *style2, style3, …, style7*. *style1* is excluded from the sort, because it equals "Container" for all 60 lineages.

Output from Step#3

oldFromName	node1	node2	node3	node4	node5	node6	node7
	≠HeaderEmphasisFixed	HeaderEmphasis	Header	HeadersAndFooters	Cell	Container	
HeaderEmphasis	=HeaderEmphasis	Header	HeadersAndFooters	Cell	Container		
Header	≠HeaderFixed	Header	HeadersAndFooters	Cell	Container		
Header	=Header	HeadersAndFooters	Cell	Container			
HeadersAndFooters	=HeadersAndFooters	Cell	Container				
Cell	=Cell	Container					
Container	≠BylineContainer	Container					

Output from Step#5

StyleNum	style1	style2	style3	style4	style5	style6	style7
1	Container	BylineContainer					
16	Container	Cell	HeadersAndFooters	Header	HeaderEmphasis	HeaderEmphasisFixed	
18	Container	Cell	HeadersAndFooters	Header	HeaderFixed		

ODS WITH PROC REPORT IS USED TO CREATE THE LINEAGE TRACER

Screen snapshots of the lineage tracer are displayed in Figures 7 and 8. The complete 61-"page" ODS|HTML tracer is included as an attachment to the paper in the NESUG Proceedings. Style elements were customized to create the lineage tracer, and the lineage tracer facilitated the customization; perfect circularity! PROC REPORT also played a central role in the tracer's development. The algorithm for inserting internal links comes from *Carpenter's Complete Guide to the SAS® Report Procedure* [2, p. 257-260].

Figure 7. Part of the cover page for the ODS tracer. Code for the three lineages with arrows is presented in Step #3 above. Pressing the green link at lineage #16 brings up a list of attributes for the member style elements displayed in Figure 8.

60 Container Lineages in the ODS Styles.Default Template
(Abstract Classes are in Blue)

Lineage#	Style#1	Style#2	Style#3	Style#4	Style#5	Style#6	Style#7
1	Container	BylineContainer					
2	Container	Cell	Data	DataEmphasis	DataEmphasisFixed		
3	Container	Cell	Data	DataEmpty			
4	Container	Cell	Data	DataFixed			
5	Container	Cell	Data	DataStrong	DataStrongFixed		
6	Container	Cell	HeadersAndFooters	Caption	AfterCaption		
7	Container	Cell	HeadersAndFooters	Caption	BeforeCaption		
8	Container	Cell	HeadersAndFooters	Footer	FooterEmphasis	FooterEmphasisFixed	
9	Container	Cell	HeadersAndFooters	Footer	FooterEmpty		
10	Container	Cell	HeadersAndFooters	Footer	FooterFixed		
11	Container	Cell	HeadersAndFooters	Footer	FooterStrong	FooterStrongFixed	
12	Container	Cell	HeadersAndFooters	Footer	RowFooter	RowFooterEmphasis	RowFooterEmphasisFixed
13	Container	Cell	HeadersAndFooters	Footer	RowFooter	RowFooterEmpty	
14	Container	Cell	HeadersAndFooters	Footer	RowFooter	RowFooterFixed	
15	Container	Cell	HeadersAndFooters	Footer	RowFooter	RowFooterStrong	RowFooterStrongFixed
16	Container	Cell	HeadersAndFooters	Header	HeaderEmphasis	HeaderEmphasisFixed	
17	Container	Cell	HeadersAndFooters	Header	HeaderEmpty		
18	Container	Cell	HeadersAndFooters	Header	HeaderFixed		

Figure 8. The drill-down from figure 7 brings up a detailed list of associated attributes for lineage #16. The attributes come from the **containerAttributes** data set described earlier in the paper. The full lineage is recapped in the title, and there is a link at the bottom of the page that takes the viewer back to the cover page.

Lineage # 16: Container Cell HeadersAndFooters Header HeaderEmphasis HeaderEmphasisFixed

Style	Default Assignment	Attribute
Container	**Abstract. Controls all container oriented elements.**	
	font = Fonts('DocFont')	FONT
	foreground = colors('docfg')	FOREGROUND
	background = colors('docbg');	BACKGROUND
Cell	**Abstract. Controls general cells.**	
HeadersAndFooters	**Abstract. Controls table headers and footers.**	
	font = fonts('HeadingFont')	FONT
	foreground = colors('headerfg')	FOREGROUND
	background = colors('headerbg');	BACKGROUND
Header	**Controls the headers of a table.**	
HeaderEmphasis	**Controls emphasized table header cells.**	
	foreground = colors('headerfgemph')	FOREGROUND
	background = colors('headerbgemph')	BACKGROUND
	font = fonts('EmphasisFont');	FONT
HeaderEmphasisFixed	**Controls emphasized table header cells. Fixed font.**	
	font = fonts('FixedEmphasisFont');	FONT
Return to Lineage List		

DATA VALIDATION

PROC SQL is applied to the **containerStyles** SAS data set created from **stylesDeftault.txt** to check for accuracy. First, a check is made to identify the style elements that inherit directly from CONTAINER:

```
proc sql noprint;
 create table chkNode2 as
  select distinct styleName
    from containerStyles
    where fromName eq 'Container'
    order by styleName;
  select count(*) into :nNode2 from chkNode2;
quit;
```

The fourteen style elements listed are identical in number and content to those in the third column on the cover page in the HTML output. Matching lineage end-style elements (or tree "leaves") is trickier and requires a sub-query:

```
proc sql noprint;
 create table chkLeaves as
  select distinct styleName
    from containerStyles
    where styleName not in
    (select fromName from containerStyles);
 select count(*) into :nLeaves from chkLeaves;
quit;
```

The NOT IN directive to the sub-query satisfies the requirement for "leaf" status: end-style-elements are never ancestors. That means they won't be listed in *fromName*. The sixty style elements produced from the query match the sixty end-names for complete lineages found in the cover page for the lineage tracer.

COMPANION PAPER

A companion paper, *Make the Most of Your Inheritance with the SAS® ODS Styles.Default Lineage Tracer* has been prepared for presentation at a future user group conference. In this paper, the lineage tracer is used to show how inheritance works in Version 9.1.3 SAS when style elements in the Styles.Default template are being altered. Along with the lineage tracer, additional font, color and attribute descriptors are employed to demonstrate that:

1) Inheritance works differently for abstract and regular style elements. Unfortunately, there is no working definition for an 'abstract' style element in SAS. The term is an Object-Oriented construct, and it refers to a base class that cannot be declared as an object. In SAS, however, abstract style elements are objects that can be changed with a REPLACE statement whereas both STYLE and REPLACE work with different results on regular style elements.

2) The FROM keyword is only needed when attribute defaults are to be transferred to an updated style element.

3) ODS inheritance is much more flexible than it is in other object-oriented programming languages such as C++. Style elements can obtain default settings from immediate and distant ancestors, themselves, and even their descendents!

4) Non-lineage attributes can also be added to a style element in a new template. Unfortunately, however, outcomes are unpredictable. An attribute may be picked up, or it can simply be ignored with no WARNING being written to the LOG. In the paper, the intractable non-lineage text justification attributes, JUST and VJUST, are reviewed in depth.

The font descriptor is also described and listed in *Macros and Conventional Macro Variables: Effective Tools for Customizing Tabular Output in SAS® ODS* [6].

SUMMARY

Tracking inheritance in the ODS Styles.Default template becomes possible when recursion is used to express 60 lineages derived from the CONTAINER style. Initially, recursion is defined and illustrated by example. Next, the data step that processes Styles.Default recursively is reviewed. After the data step review, several screen snapshots showing portions of the final output produced in ODS with an HTML destination are presented. Finally, PROC SQL is used to check the HTML output for accuracy. A section that describes the companion paper *Make the Most of Your Inheritance with the SAS® ODS Styles.Default Lineage Tracer* has also been added to show why it is a good idea to have the lineage tracer on your desktop when you need to change the definition for a style element.

COPYRIGHT STATEMENT

use, or quote text from the paper in any type of publication you are welcome to do so. All you need to do is to cite the paper. For all uses that result in corporate or individual profit, written permission must be obtained from the author. Conditions for usage have been modified from http://www.whatiscopyright.org.

REFERENCES

[1] Carpenter, Art. *Carpenter's Complete Guide to the SAS® Macro Language: Second Edition.* Cary, NC: SAS Institute Inc., 2004.

[2] Carpenter, Art. *Carpenter's Complete Guide to the SAS® REPORT Procedure.* Cary, NC: SAS Institute Inc., 2007.

[3] Koch, George and Kevin Loney. *ORACLE: The Complete Reference, Third Edition.* Berkeley, CA: Osborne McGraw-Hill, 1995.

[4] Tanenbaum, Aaron M. and Moshe J. Augenstein. *Data Structures Using Pascal.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[5] Watts, Perry and Samuel Litwin. *Using SAS/GRAPH® Software to Create Trees that Model Developmental Biological Phenomena.* Proceedings of the Sas Users Group International Sixteenth Annual Conference. SAS Institute, Inc. Cary, NC, pp. 779-784, 1991.

[6] Watts, Perry. *Macros and Conventional Macro Variables: Effective Tools for Customizing Tabular Output in SAS® ODS*. Proceedings of the 23rd Annual Northeast SAS Users Group Conference. Baltimore, MD 2010, paper #CC32.

WEB CITATIONS

[7] http://en.wikipedia.org/wiki/Recursion_(computer_science). *Recursion (Computer Science): From Wikipedia, the free encyclopedia*.

SAS PAPERS ON RECURSION (from *Carpenter's Complete Guide to the SAS® Macro Language* [1, p.394])

Adams, John H. *The Power of Recursive SAS Macros: How Can a Simple Macro do So Much?* Proceedings of the Twenty-Eighth SAS® User Group International Conference, Seattle, WA, 2003, paper #136.

Benjamin, William E. Jr. *A Pseudo-Recursive SAS Macro*. 1999. Online article available at www.sas.com/techsup/download/observations/obswww18/obswww18.pdf.

Rhoades, Stephen. *Recursion? SAS? Let's Fake It.* Proceedings of the Fourteenth Annual NorthEast SAS® Users Group Conference, Baltimore, MD, pp. 652-657, 2001, paper #PS8013.

Ward, David L. *Using Recursion in the SAS System.* Proceedings of the Pharmaceutical SAS Users Group Conference, Boston, MA, pp.97-98, 2001, paper # CC12.

ACKNOWLEDGEMENTS

The author thanks Stan Legum, NESUG section co-chair, for his thoughtful review of her manuscript.

WHAT'S IN THE NESUG 2010 PROCEEDINGS OR AVAILABLE BY REQUEST

The ODS lineage tracer is available for download from the ZIP file associated with this paper. Additional source code is available by request.

TRADEMARK CITATION

CONTACT INFORMATION

The author welcomes feedback and requests for source code via email at perryWatts@comcast.net.
Related papers and Zip files can also be found at http://www.screencast.com/users/PerryWatts