SAS® GLOBAL FORUM 2020 #SASGF

Paper SAS4454-2020

CASL, a Language Specifically Designed for Interacting with SAS® Viya®

Jerry Pendergrass, SAS Institute Inc.

ABSTRACT

CASL is a new language designed to run SAS® Cloud Analytic Services (CAS) actions and process responses to generate a report. To make it easy for SAS users, the language syntax mimics the syntax of the DATA step. CASL is not just a language, but a programming environment that is embeddable into any program. CASL is available through the CAS procedure or as the action runCasl in the CASL Server action set. Learn how to use the CASL language to pipeline actions one after another to produce a report. You will learn how to use CASL to create your own result tables from action results. Allow me to introduce you to a very powerful language that is simple to use.

INTRODUCTION

CASL is a powerful language for running actions in CAS. CASL is embedded in three products in SAS. These include a procedure that runs within SAS and actions that run on the server. CASL has one goal in mind, and that is to allow you to run actions in CAS and to process the results of those actions into useful data and sophisticated reports.

CASL IS EMBEDDED IN THREE SAS PRODUCTS

CASL is a language environment that can be embedded into any program or environment. CASL is embedded in these SAS products:

- PROC CAS
- The sccasl.runcasl action
- The builtins.defineActionSet action

PROC CAS

The CAS procedure allows a user to submit programs to run actions from within SAS. Since it is a procedure, it has access to typical facilities within SAS including macros, libnames, filenames, options, and the output delivery system (ODS). The only purpose of PROC CAS is to run and process CAS actions. PROC CAS can be used as an interactive procedure, where code is run at each RUN statement or as a batch procedure where the code is submitted all at once. PROC CAS handles task interruption (^c), which stops the current execution of CASL code and returns control back to the procedure. To exit the procedure, either start a new DATA step/ procedure or enter `quit;`. Any outstanding CASL code is executed before PROC CAS exits.

Here is simple syntax that prints the results from fetching a result table:

```
carstbl = {name="carssashelp", where='Buick' = make"};

table.fetch /
table = carstbl,
fetchvars = {"make", "model", "type", "msrp"}, sortby={name="msrp"};
```

You can use variables to construct parameters, or you can create them as expressions on the line that executes the action. The syntax for creating parameters for an action will be explained later.

THE SCCASL ACTION SET

The sccasl action set implements a CAS action named 'runcasl', which executes a CASL program in the server environment. The user may specify global variables to be initialized before the program starts running. The CASL program might return any number of results and might provide an exit status. Any log messages are sent back to the user asynchronously.

The syntax for the Runcasl actions is:

```
sscasl.runcasl code=<code string>  vars={name1=value1, name2=value2,... }
                                   macvars={macname1='macvalue1',
                                         macname2='macvalue2',...};
```

The results of Runcasl are determined by the CASL program. The function 'send_response' is called to send back a response. A response consists of one or more directories pass to the function 'send_response'. You can call 'send_response' as many times as you like. The 'exit' function is called to send back the status from this action and terminate execution of the action.

Below is a simple example that obtains a record count (in variable recordCount) from a table and returns a summary of the resulting table.

If an error is detected, the exit status for the action is specified.

```
source codeReccount;
        table.recordCount result=count status=s1 / table=myname;
        print s1;
        if s1.severity != 0 then do;
                exit(s1);
        end;
```

```
          simple.summary result=mysum status=s / table=myname;
          if s.severity != 0 then do;
                  exit({severity=5,reason=5,statusCode=5});
          end;
          send_response({table=mysum});run;
  endsource;
  sscasl.runcasl code=codeReccount vars={myname='cars'}; run;
  run;
```

Note that any CAS client can submit a CASL program to execute the Runcasl action. Here is the same example using the Python client.

```
  Res.status = s.sccasl.runcasl(code=' table.recordCount result=count
status=s1 /
                                                      table=myname;
                                       print s1;
                                       if (s1.severity != 0)
                                           then  exit(s1);
                                       simple.summary result=mysum status=s /
                                                      table=myname;
                                        if (s.severity != 0) then

exit({severity=5,reason=5,statusCode=0});
                                        send_response({table=mysum});';
```

THE DEFINEACTIONSET ACTION

CASL is embedded in the built-in action defineActionSet that allows the user to create their own action set using CASL as the programming language for this action. You can write your own actions using CASL syntax. For example, you could write actions to load all the tables in a directory hierarchy, save all the tables in a caslib, or group several action invocations into a macro-like module to share with other developers. The defineActionSet action provides a wrapper on top of the Runcasl action interface for an action, just like other actions in CAS. The user will see no difference between an action created with defineActionSet and an action that is deployed with CAS.

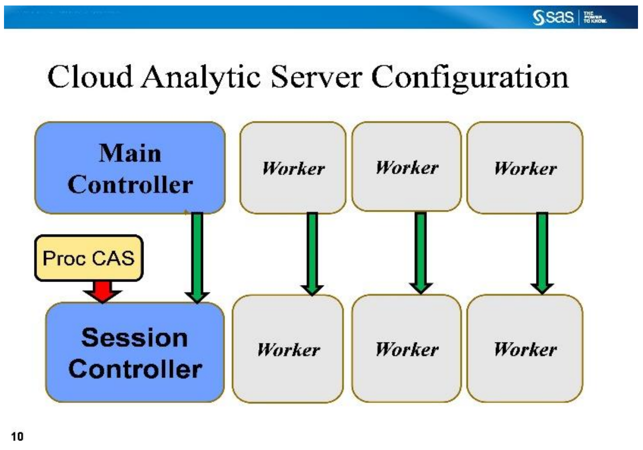
Below is the definition of the codeReccount action using the same CASL code in the previous examples.

```
 builtins.defineActionSet /
   name="myActionSet"
   actions={
      {
        name="codeReccount"
        desc="summary on record count"
        parms = {
           { name="myname" type="string" required=TRUE}
        }
        definition=codeReccount;
      }
   };

 myActionSet.codeReccount result=sum   myname=cars; run;
```

USING CASL TO RUN A CAS ACTION

Actions define work to be executed on the CAS server. Each action has a name, action set, and parameters in the form of a directory. A picture of the interaction between CASL and CAS is shown below:



One of the key purposes for using CASL is to prepare arguments for the execution of an action. An action is submitted by specifying the action name or actionset.name, followed by configuration parameters that help define how the action is to be processed. You then provide keyword arguments that are used to process the action. The arguments are specified as a set of directory entries. It is the specification of this directory structure that is a focus of CASL. Two methods allow CASL to provide arguments for a CASL action:

- Specification of an array
- Specification of a directory

CREATION OF AN ARRAY

An array is a list of values that are referenced using an index instead of a named key. A list is created either by using braces or brackets.

```
      fetchvars = {"make", "model", "type", "msrp"};
```

This example creates an array of 4 items. Indexes start at 1 and increase. A value at a given index does not have to exist, but if an undefined array entry is referenced, a missing value will be used. Arrays may contain as many dimensions as are needed, separated by commas. You may also specify an array as 4 assignments.

```
    fetchvars[1] = “make”;
    fetchvars[2] = “model”;
    fetchvars[3] = “type”;
    fetchvars[4] = “msrp”;
```

CREATION OF A DICTIONARY

A dictionary is a list of values with a different named key for each value. The order of the list is not guaranteed. You may traverse through a dictionary by referencing the values one at a time in a loop. Let’s look at syntax for creating a dictionary as parameters to an action. Suppose you want to call the Fetch action to download a CAS table. The name of the table is Carssashelp. I want to apply a WHERE clause to the table and I want to specify which variables I want in this download.

You can set up the table parameter as either:

```
  carstbl.name  = "carssashelp";
  carstbl.where = "'Buick' = make";
```

or as:

```
 carstbl = {name="carssashelp", where='Buick' = ‘make’};
```

This second assignment group is the typical method used to set up an action parameter. It is more intuitive.

Note that in the syntax below, I use the Carstbl variable for the table parameter and then specify the variables to fetch as an array of strings. I then added in a sort specification.

```
  table.fetch /
    table = carstbl,
    fetchvars = {"make", "model", "type", "msrp"}, sortby={name="msrp"};
```

The default behavior when executing an action is to print the results. The user may specify a variable to receive these results. I have changed the action above to place the results into the dictionary named Tab. Note that results are always a dictionary of values. Those values might be doubles, strings, integers, or arrays and other dictionaries. The action defines the form of the results sent from that action.

```
table.fetch result=tab /
   table = carstbl,
   fetchvars = {"make", "model", "type", "msrp"}, sortby={name="msrp"};
```

USING CASL TO PROCESS THE ACTION RESULTS

What does it mean to get results from an action? The results are returned in the form of a dictionary. Each client presents these results to the user in the context that makes the most sense for that client. The context that makes sense for CASL is as CASL variables, thereby losing no data in the translation. The result values are organized into dictionaries and arrays. The supported data types include double, integer, utf8 string, Boolean, time data, datetime, varbinary, and a result table.

The result table is the most common result type. A result table is a data table with rows and columns. Each column has a name, label, and format. CASL uses the column name as a string to reference a specific column. CASL represents a result table as a two-dimensional table, where the 1st dimension references the row and the 2nd dimension references the column. Columns can be referenced as either by name or by index.

It is very important to understand that a dictionary is always returned from an action. This dictionary might contain a result table. To access the table, you must reference the dictionary entry that contains the result table. Suppose I fetch a table using this syntax:

```
fetch result=tab table={name="cars"} to=20;
```

The variable 'tab' is a dictionary, not the result table. To get the result table you must reference the dictionary entry.

```
mytab = tab.fetch;
```

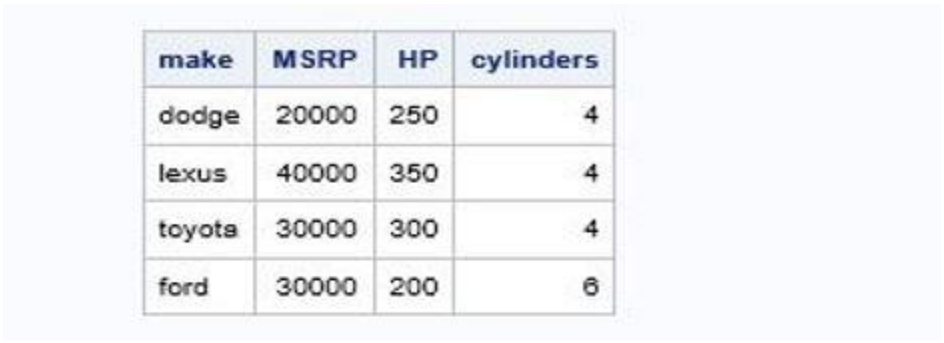
CASL is designed to formulate results into new parameters and elegant reports. CASL supports a comprehensive expression parser to manipulate results. CASL is not object-oriented, but does have internal Class variables such as arrays, dictionaries, and result tables. The primary operator in CASL is the DOT ('.') operator. This operator is polymorphic, meaning that the operation depends on the data type of two operands. If the 1st operand is a dictionary, then the second operand is either an index in the dictionary list or a key used to look up the value. CASL also supports a large resource of functions to operate on numbers, strings, formats, or result tables.

NEWTABLE AND ADDROW FUNCTIONS

CASL allows you to take data from anywhere and create a result table using that data. The 'newtable' function creates a result table with the parameters specifying the name of the table, the name of the columns, and the data type of the columns. Additional parameters might specify rows of data to add to the result table. You can create the table as a blank table and add rows later using the ‘addrow’ function. This gives you the power to combine data from anywhere into a table that is a compatible CAS result table. Here is an example of creating a result table.

```
   t=newtable( "table", { "make", "MSRP", "HP", "cylinders" },
     { "varchar", "int64", "integer", "int64"},
     {"dodge", 20000, 250, 4},
     {"lexus", 40000, 350, 4});

   addrow(t,  {"toyota", 30000, 300, 4},
                 {"ford  ", 30000, 200, 6} );
   Print t;
```



make	MSRP	HP	cylinders
dodge	20000	250	4
lexus	40000	350	4
toyota	30000	300	4
ford	30000	200	6

WHERE CLAUSE AND COMPUTE OPERATOR

CASL supports two special operators for result tables. The WHERE clause allows you to select rows from the table based on expression evaluation. The variables for each row are available as normal variables in the expression. An expression is evaluated, and if the result is true, then the row is added to the new result table.

Let's use the Cars table as an example. This table contains a column named MSRP. I can use the WHERE clause to select all rows where MSRP is greater than 30000. The expression can use any available CASL variable, not just variables from the result table.

```
newcar = cars.where( msrp > 30000);
```

The Compute operator allows you to create a new column. You supply the name, label, and format for the new column as the 1st argument. The second argument is an expression used to calculate the value for each row. As with the WHERE clause, the values of the result table variables will change to match those of the row being processed. The data type is defined by the 1st expression evaluated.

```
newcol = cars.compute( {"ratio", "msrp/invoice",best5.3}, msrp/invoice);
```

Here is an example using the Cars data set. Subset the result table using WHERE and Compute, and then reduce to 5 rows and 4 columns.

```
  res  = cars.where(Horsepower<200).compute({"dphp","Dollar/HP",DOLLAR8.},
                                           invoice/Horsepower)
                                   [1:5,${Model Horsepower invoice dphp}];
```

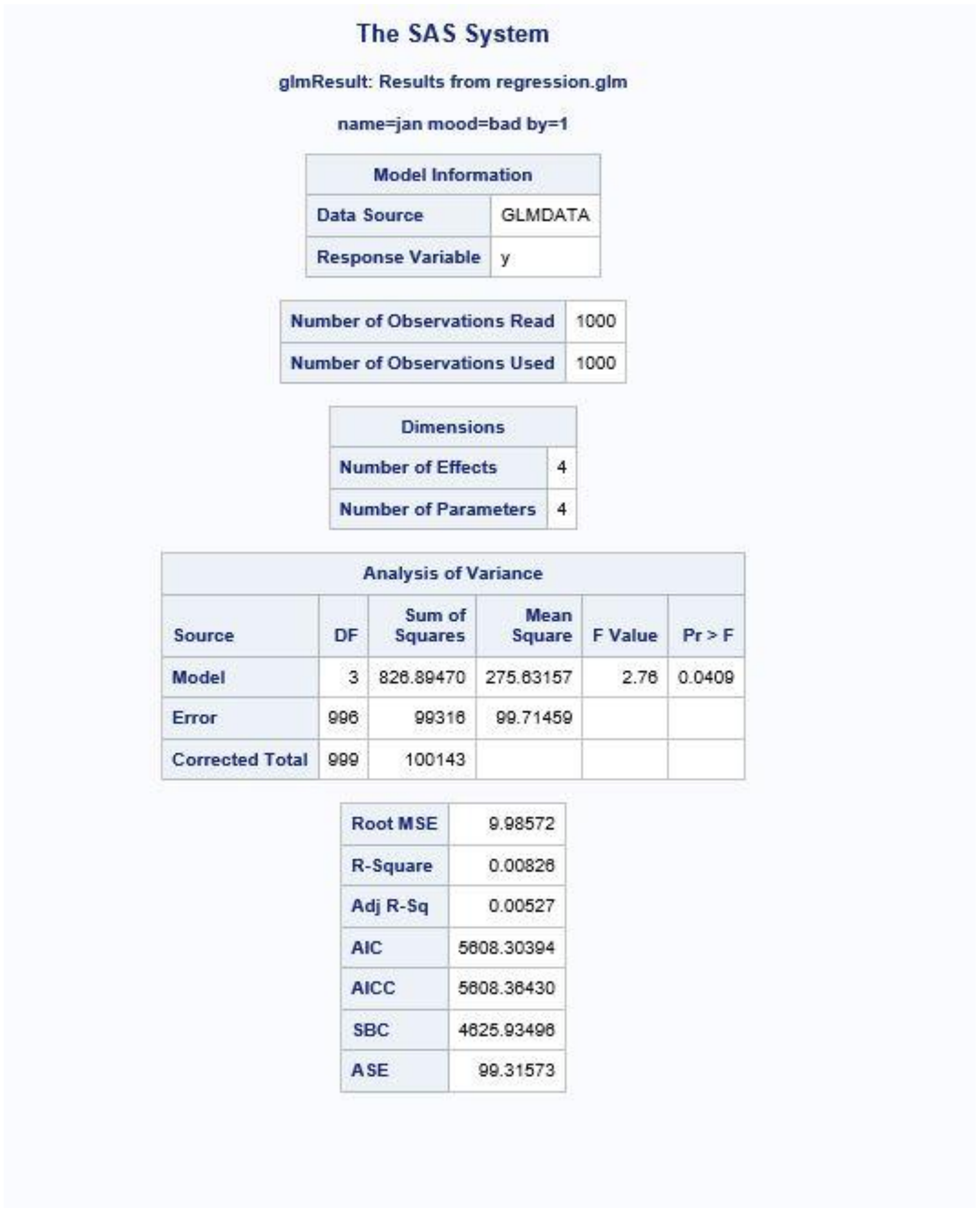
PRINTING A REPORT USING THE OUTPUT DELIVERY SYSTEM

The Output Delivery System (ODS) allows CASL to display the contents of your result tables based on the intent of a given action. When CASL is used to print a result table, ODS is called to display the result. For example, suppose I run GLM on a given dataset:

```
  proc cas;
   loadactionset "regression";
        glm result=glmResult
            table={name='glmdata', groupBy={'name', 'mood', 'by'}}
          model={depvars={{name='y'}}
          effects={'x1', 'x2', 'x3'}};
     if (_status.severity == 0)
           print   glmResult[2:6];
```

In the example above, I requested to group values by specified variables for the regression. Using the describe statement, I found out that the 1st group-by result is in indexes 2-6.

Note the check for a good return status validates that the data I expect has been returned. The ODS output is presented below:



The SAS System

glmResult: Results from regression.glm

name=jan mood=bad by=1

Model Information	
Data Source	GLMDATA
Response Variable	y

Number of Observations Read	1000
Number of Observations Used	1000

Dimensions	
Number of Effects	4
Number of Parameters	4

Analysis of Variance					
Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	3	826.89470	275.63157	2.76	0.0409
Error	996	99316	99.71459		
Corrected Total	999	100143			

Root MSE	9.98572
R-Square	0.00826
Adj R-Sq	0.00527
AIC	5608.30394
AICC	5608.36430
SBC	4625.93496
ASE	99.31573

USING FORMATS TO ENHANCE YOUR REPORTS

CASL supports all the threaded kernel stand-alone formats and user-defined formats. A format is represented in CASL using SAS syntax.

```
<name><width>.<decimal>
```

If a variable is assigned to a format, that variable can be used as a format. This allows the user to create dynamic formats. The Put function operates the same as the Put function in the DATA step.

```
   <string> = put(<expression>, format);
```

The expression is evaluated and then converted to a string using the supplied format. It is useful to use the Put function with the PRINT statement.

```
        print  "the value is " put(1.234678, d4.2);run;
```

The resulting value is 1.23.

CONCLUSION

As can be seen, CASL is a powerful language that allows you to run actions and process the results. The expression parser provides a rich syntax to create parameters for actions and produce elegant reports. ODS is fully integrated into CASL, providing the same level of output as you expect from a procedure. The ability to leverage the CASL program language in three separate products gives you the flexibility to re-use code and deploy an application easily into the CAS Server. The next time you want to run an action against the server, try PROC CAS or a CASL action. See how easy it can be.

RECOMMENDED READING

- Getting Started with CASL: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=casl&docsetTarget=titlepage.htm
- CASL Reference: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=proccas&docsetTarget=titlepage.htm
- CASL Programmer's Guide: https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=caslpg&docsetTarget=titlepage.htm
-

CONTACT INFORMATION

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

Jerry Pendergrass
SAS Institute Inc
919-531-7766
jerry.pendergrass@sas.com