Paper SAS3177-2019

Got Results? Let CASL Help Turn Them into Superior Reports

Jerry C. Pendergrass, SAS Institute Inc.

ABSTRACT

CASL is a new programming language that simplifies running actions in SAS® Cloud Analytic Services and processing the results of those actions. A core strength of CASL is the ability to receive the results of an action (tables, dictionaries, arrays, and scalar values) and transform those results into a final report. You can combine CASL's expressiveness for transforming results with the capabilities of SAS Output Delivery Service (ODS) to produce refined reports. We present CASL syntax to extract rows and columns using a WHERE clause and column subsetting operators. Find out how to pipeline actions using CASL to prepare arguments for an action using the results of a previous action. Come and see CASL in action producing polished report

INTRODUCTION

The Cloud Analytic Server (CAS) returns action results as a dictionary. This dictionary contains a CAS value for each unique key. The client imports these results in the form the user expects, fitting the data model of the client. The client I am presenting in this paper is the CASL client. CASL is a new language designed to create parameters for an action and receive the results in an untouched from, allowing the user to process, manipulate, and design reports from these raw results. CASL's native variable types support all the native types found in the results returned from CAS, enabling CASL to present the results untouched for processing the results into a finished report. CASL is built with one goal mind: Allow you to submit actions to the server and receive, process, manipulate, save, or display the results from that action.

CASL provides an interface embeddable into any program. The embeddable approach allows CASL to be independent of any specific system and allows CASL to deploy into many different environments. The procedure CAS provides the environment in SAS® to run CASL programs. This is the platform used to describe how to use CASL to produce reports. PROC CAS allows CASL to access the features of the SAS system, including macro, ODS and SAS® Studio.

The variables in CASL are weakly typed. You do not declare a variable, it gets its data type from the expression assigned to it. This allows the action results to easily to placed into the CASL environment untouched. The path to understanding how to process results into a better form requires you to understand who CASL operates and how the operators, functions, and statement are used to achieve a result. The CASL reference and programmers guide is a good start to understand the syntax and operators supported in CASL. Let's see how CASL allows a user to process results.

WHAT ARE RESULTS?

What does it mean to get results from an action? These results are returned in the form of a dictionary. The key, which is unique, is the name of the value. 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 data types include double, integer, utf8 string, Boolean, time, data, datetime, varbinary, and result table.

The result table is the most common result type. A result table is a data table with rows and columns. Each column as 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. If 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;
```

Now 'mytab' is a variable of type result table.

REFERENCING VALUES IN A RESULT TABLE

Let's look at an example using the cars data set. I can reference the 'msrp' column for row 1 by using this syntax:

```
msrp = cars[1,"msrp"];
```

This will assign the value of msrp for row one to the variable 'msrp'.

You can extract a specific row into a dictionary by selecting a row and not specifying a column.

```
carsrow =  casrs[5];
```

This extracts row five into a dictionary:

```
{_Index_=5,Make=Acura ,Model= 3.5 RL 4dr, Type=Sedan ,Origin=Asia,
 DriveTrain=Front, MSRP=43755, Invoice=39014, EngineSize=3.5, Cylinders=6,
 Horsepower=225,  MPG_City=18, MPG_Highway=24, Weight=3880, Wheelbase=115,
 Length=197}
```

If you want to print values from this dictionary, use the 'put' function to format the values.

```
  print  carsrow.Make "  " carsrow.Model " " put(carsrow.msrp,  DOLLAR8.) "  "
                                                put(carsrow.mpg,d5.2)
```

```
   Acura           3.5 RL 4dr           MSRP= $43,755   MPG_City= 18.0
```

You can also extract a column into an array using this syntax:

```
      carcol = cars[,"Model"];
```

'carcol' is now an array containing the models, one for each row. Note the comma next to the left bracket means to take all of the rows.

SUBSETTING A RESULT TABLE

CASL support multiple methods to subset result tables. The best form for generating a report is a result table. The results you receive from an action contains the raw data you need to produce a final report. The ability to move and select columns and rows is important. The ability to create a new result table from the data you have will allow you to combine data from multiple sources. CASL provides many functions and operators to allow you to reconfigure and generate your final report.

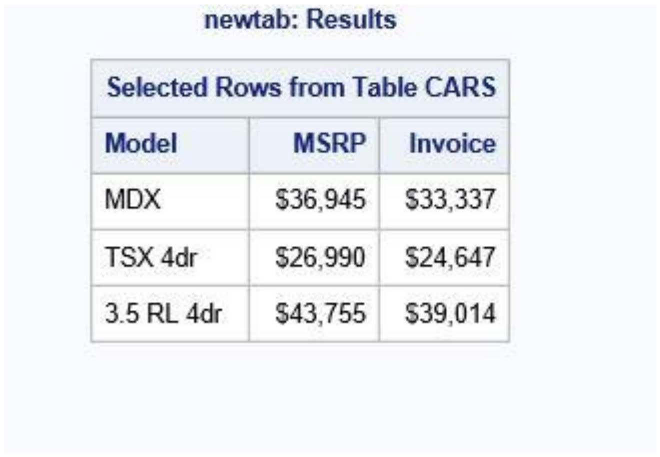
Row and Column Sub-Setting Operators

CASL allows you to supply an array of values as the indexes to the row and column dimensions. This allows you to re-design a result table column by column and row by row.

If a list is given as the index to a row numbers, that list specifies the rows to keep. If a list of columns is given for the column dimension, that defines the order and list of columns to keep.

Let's look at an example:

```
   Fetch result=tab   table={name="cars"} ;
    newtab = tab.fetch[{1,3,5},{"msrp","invoice","model"}];
    print newtab;
```


newtab: Results

Selected Rows from Table CARS		
Model	MSRP	Invoice
MDX	$36,945	$33,337
TSX 4dr	$26,990	$24,647
3.5 RL 4dr	$43,755	$39,014

Where and Compute Operators

CASL supports two special operators for result tables. The WHERE operator allows you to select rows from the table based on an expression evaluation. The variables for each row are available as normal variables. An expression is evaluated and if they result is TRUE 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 operator 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);
```

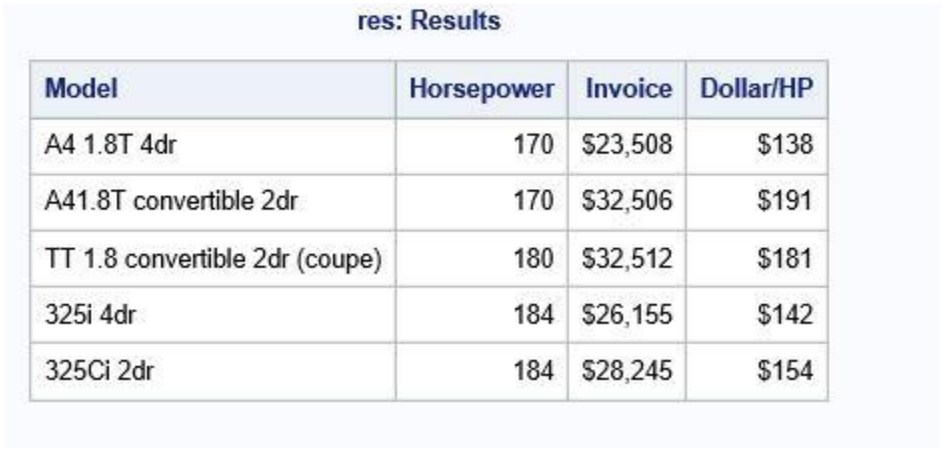
The 'compute' operator allows you to create a new column. You supply the name, label, and format for the new column as the 1st argument. The second argument is an expression used to calculate the value for each row. As with the WHERE operator, the values of the result table variables will change to match 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 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}];
```

```
print res;
```



res: Results

Model	Horsepower	Invoice	Dollar/HP
A4 1.8T 4dr	170	$23,508	$138
A41.8T convertible 2dr	170	$32,506	$191
TT 1.8 convertible 2dr (coupe)	180	$32,512	$181
325i 4dr	184	$26,155	$142
325Ci 2dr	184	$28,245	$154

Comparison Operators

CASL supports all of the standard comparison operators. A comparison operator can be used on any data type. If used on an array, the result will be an array containing a list of the indexes matching the comparison. These indexes can then be used to access the values in the array. For example:

```
list = {1,5,2,8};
subset = list[list>4]
```

The operation 'list>4' produces {2,4}. This is the list of indexes that are greater than 4. The operation 'list[{2,4}]' selects the values at index 2 and 4 and returns a list with two values {5,8}.

The 'shorter' operators that use a ':' after the operator compare using the shorter of the two lengths. So, if one string is 10 bytes long and the other is 5 bytes long, the '=:' only compares the 1st 5 bytes to make the determination of equality.

You many compare one directory to another directory. The order of the values in the directory does not matter as a long as both have the same values and keys. Using the greater than or less than operators on some of the data types can give nonsensical answers. Comparisons for equality can compare values of different data types. A string is equal to a varbinary if the contents are equal.

Using sub-setting Operators on Arrays and Result Tables.

Let's look closer at subsetting an array or result table. Subsetting allows you to reorder, columns, reduce or add columns, or select which rows are to be kept. Let's look at some more examples:

```
Given:  x = {1, 4, 8, 2, 9,3};
```

```
y = x[(x>3) && (x < 9)];
```

This expression has many parts. The (x > 3) recognizes that x is an array. The indexes of the values that match the conditional are placed in a list, {2, 3, 5}. The next conditional is (x < 9). That results in {1,2,3,4,6}. These two lists are ANDed together resulting in {2,3}. This index list is evaluated against x and the indexes 2 and 3 are selected. The result is {4,8}. So, if I print x [{2,3}], I get {4,8}. The list, in the brackets, enable you to select which items are kept.

I can also assign multiple values at the same time.

```
x[(x>3) && (x < 9)] = 3.14;
```

Once again, the list within the brackets selects the values to be set. This sets the second and third values to 3.14. This works for multi-dimensionial arrays are well. Here is an example.

```
x[1:5,2:6]  = 1;
```

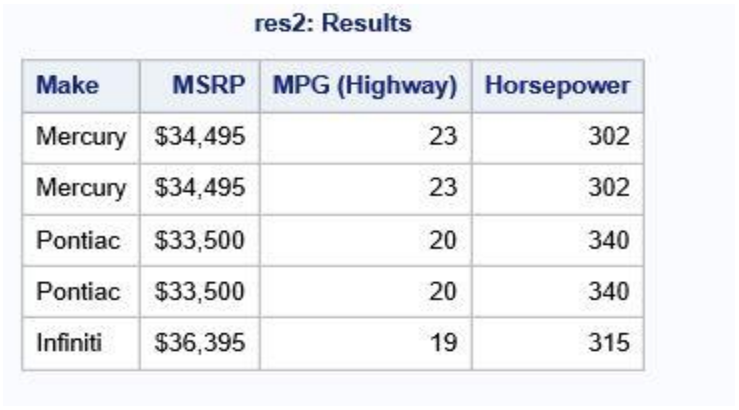
This initializes a two-dimensional array with rows 1 through 5 and columns 2 through 6 being set to 1. This is much easier than running two loops to initialize these values. You should be cautious about assuming this assignment we initialize x to a scalar value before initializing the array. If this line followed the previous line, this would update the values the previous array. Setting a value to a scalar value such as 1 will make sure the value is initialized the way you want.

CREATING A REPORT FROM TWO RESULT TABLES

Let's consider a problem where you have two tables with the same columns but different types of data. One has data on inexpensive cars and the other on expensive cars. I want to fetch the data and apply operators to combine the two tables giving a specific criterion for keeping rows from each table. I then want to sort the resulting table and print the 1st 5 rows.

```
fetch r=tab  table={name="cars"} to=500;
fetch r=etab table={name="expensivecars"} to=500;
res   = tab.fetch.where((msrp>20000) && (msrp < 40000));
res1 = etab.fetch.where((msrp<40000) && (msrp > 20000));
res2 = combine_tables({res,res1})
                [,{"Make","MSRP","MPG_Highway", "Horsepower"}];
res2 = sort_rev(res2.where( Horsepower>300), "MPG_Highway")[1:5];
print res2;
```

This code fetches both tables from the server. It uses the WHERE operators to select rows from these two tables, and then combines these rows into one table. The 'combine_tables' is a very useful function, especially when dealing with GROUPBY results. The resulting table's columns are subsetted to 4 columns. This new result table is subsetted to only include rows where horsepower is greater than 300 hp and then sorted high to low using MPG_highway as the sort key. This sorted table is reduced to the 1st 5 rows. The ':' operator is used to create a list {1,2,3,4,5}. The resulting report is displayed using ODS.


res2: Results

Make	MSRP	MPG (Highway)	Horsepower
Mercury	$34,495	23	302
Mercury	$34,495	23	302
Pontiac	$33,500	20	340
Pontiac	$33,500	20	340
Infiniti	$36,395	19	315

WHAT OTHER VARIABLE TYPES DOES CASL SUPPORT

The data types for CASL variables includes all the data types for a result table and data types for possible results. CASL also support additional data types used to support processing action results. Let’s consider those additional data types.

FUNCTION AND FORMAT VARIABLES

A CASL variable can contain the location of a function. This variable can be used to call the function it references. You can overwrite a static function by redefining it as a function variable. A format variable is created from a format definition. This variable can be used in place of a format name.

Here is an example:

```
function mysquare( val );
   Return(val * val );
end;
square = mysquare;
sq2 = square(2);

myfmt = best5.2;
put(3.1415,myfmt);
3.14
```

MISSING VALUES

CASL supports all 27 missing values from _. to .z. All the missing values returned by the server are supported. The default missing value is . (dot). Missing values are designed to sort as the smallest values where .z is the smallest. Missing values are referenced CASL using this syntax:

```
._ . .a .b  .... .z
```

DICTIONARIES

Dictionaries are a key data type in CASL. Parameters to actions and results from actions are represented as dictionaries. CASL supports many operators and functions that operate on a dictionary. A dictionary is a list of values where a unique name is associated with each value. A directory might contain directories and arrays. The depth of a directory or array is unlimited. A directory is referenced using the DOT operator or using brackets []. The order of values in a dictionary might change. Let's look at some dictionary examples:

I have a dictionary named 'results', which has a value named 'summary'. You reference 'summary' as either:

```
  value = results.summary;
  value = results["summary"];
```

Using the brackets allows the user to specify an expression for the name of the entry in the dictionary. The bracket method allows the dictionary key to contain spaces and special characters or any expression.

If a dictionary contains another dictionary, then reference the second dictionary using the same DOT or bracket syntax.

```
  value = results.summary.list.number;
```

A dictionary value is set by either referencing the dictionary entry and assigning it a value or by using the braces syntax to assign a directory structure to a variable. To create a dictionary named 'a' that contains a dictionary 'b' that contains an integer named 'c', you use this syntax:

```
  a.b.c = 1;
```

The assignments below create a dictionary with 5 entries. The 4th entry is an array. The 1st example uses the using {} to create the directory. The second example creates the same variable but using the DOT syntax to reference each variable. The braces are a more elegant method to creating a dictionary.

```
   arg = {arg1=1,arg2=3,arg3=4, arg4={2,4}, arg5="abc"};

   arg.arg1 = 1;
   arg.arg2 = 3;
   arg.arg3 = 4;
   arg.arg4[1] = 2;
   arg.arg4[2] = 4;
   arg.arg5 = "abc";
```

The DESCRIBE Statement

The DESCRIBE CASL statement displays the structure of an expression. The DESCRIBE statement details the directory or array structure along with values in the dictionary or array. This recursively traverses all values in the expression. You might appreciate the DESCRIBE statement when you receive a result from an action and need to know the structure of that result. The statement details the names and data types of an action result. The statement can be used on a result table to get the structure of the table. For example, this is the output of describing the variable x.

```
  x.y[2].z.a["abc"].t[10] = 1;

   Dictionary ( 1 entries, 1 used);
    [y] Array ( 2 entries, 1 used);
     [ 2] Dictionary ( 1 entries, 1 used);
      [z] Dictionary ( 1 entries, 1 used);
       [a] Dictionary ( 1 entries, 1 used);
        [abc] Dictionary ( 1 entries, 1 used);
         [t] Array ( 10 entries, 1 used);
          [10] int64_t;
```

As expected, 'x' is a dictionary containing a value 'y'. 'y' is an array with the current size of two values, one of which is initialized. The initialized value is a dictionary, which contains a dictionary with the name 'z'. 'z' is a dictionary containing a dictionary named 'abc'. ABC is a dictionary containing an array named 't', where one out of 10 values is initialized. The 10th value is initialized as an int64.

Note that the name of the variable is case insensitive. The reference to the dictionary 'abc' could have been a.abc.t[10]. The bracket quotation mark syntax is equivalent to the DOT operator syntax, when operating on a dictionary. The brace syntax is a great syntax to create parameters for an action.

ARRAYS

An array is a list of values without keys, where the 1st index to the array is 1. The array can be a sparse array with only certain values assigned. Any value that is not assigned will return the value of missing (.). Arrays might contain arrays or dictionaries, providing the support for multi-dimensional arrays. The depth of multidimension arrays in unlimited. An array item is referenced using brackets. Multi-dimensional arrays use commas to separate the different dimensions. For example:

```
        x =  value[3];
        x =  value[3,4*2];
```

An array can contain a dictionary. To reference a dictionary in an array, use this syntax.

```
x = value[3].subvalue.value1[index*3];
```

This syntax takes the third array value in 'value', which is a dictionary and references the key 'subvalue'; This value is a dictionary and I reference an array named 'value1' and take the index 'index" times 3. Arrays and dictionaries are key to initializing action parameters. For example, the variable ‘by' will be used to execute an action:

```
     by = {groupBy={"OCCUPATION", "car",
                    "gender", "education"},
           groupByColors={"red","green","white","blue"}};
```

Note that `${red green white blue}` can be substituted for `{"red","green","white","blue"}`. The $ in front of the braces means the contents in the braces are constant strings. The value of the variable 'by, is a dictionary. It contains two values GROUPBY and ‘groupbybolors'. Both are arrays containing strings. As shown earlier, this array can be created by setting each individual value in the array. In practice using the braces to create dictionaries and array will be much easier and produce much simpler code. Once an array or dictionary has been created, you can reference individual values as needed.

The operators supported in the expression parser allow for easy subsetting, combining, and comparing of arrays and dictionaries. The ability to manipulate arrays, dictionaries, and result tables in the expression parser is the key to generating a well-designed finished report.

THE EXPRESSION PARSER

In the examples so far, the expression parser has been used to resolve the value of an expression. The only method to manipulate a value in CASL is through the expression parser. A unary operator operates on a single value resulting in a single value, A binary operator operates on two values resulting in one value. The parser is a push down stack with precedence. The precedence controls when values are popped, and operations occur. Functions calls occur only in the expression parser.

CASL is a weakly typed language, the context of an expression is only known once an expression is evaluated. This means execution errors might occur if the type of an operand is incorrect for the operator being used. An error in evaluation normally results in a missing value.

The operators in CASL are polymorphic. This means the operation between two values depends on the types of those values.

- The plus operation between two integers results in the addition of the two values.

- The plus operation between an array and a value results in the value being added to the array.
- The array index operation on a result table indexes into the table as if it were a two-dimensional array.

For example:

```
date_value = table[1,"date"];
```

This will return the value at row 1, column = "date". The expression does not know the type of the operand at compile time. It is not until the operation is executed that is understands that the array is a result table. If an error is detected, the result is set to missing.

It is important to understand the precedence of each operators. This precedence controls when an operation will be performed and queued on a stack to be executed later. This is the same concept that most languages have to control the order of operations.

CASL Functions

CASL supports many function types. The type defines the purpose of the function, why it was written, and can it be overridden using a function variable. The precedence of a function is just below that of the DOT operator. Below is a list of function types:

- User functions written in the CASL language
- Utility run-time functions
- Application supplied functions
- DS2/DATA STEP functions

Useful Utility functions

CASL provides many functions to process results from an action. Below are some of the more useful functions for processing a CAS results into a finished report.

Return the UUID of the specified session.

```
string   uuid(<session name>);
```

Format the expression using the supplied format.

```
string   put(<value>, <format>);
```

Determine if an expression has the specified type.

```
Boolean isType(<value>, <datatype>);
```

Create a new result table.

```
table newtable(<string>, {labels}, {data types}, {row1}, {row2}, ...};
```

Return a dictionary where keys are the column name and values are the column label.

```
table     tabcolumns(<table>,column);
```

Return an array of strings describing the data type for each column.

```
table     tabtypes(<table>,{types});
```

Add one or more rows to a supplied result table.

```
table     addrow(<table>, {row}, …);
```

Return the 1st dictionary in the expression.

```
table     findtable(<value);
```

Concatenate the list of result tables in to one table.

```
table         combine_tables({list of tables},<name to match>);
```

Return the contents of the supplied file. String is the default type.

```
string     readpath(<pathname>,<"table", "binary","string">);
```

Determine if the given key is in the dictionary.

```
Boolean  exists(<dictionary>,<key>);
```

Sort the array or table in ascending order. Use sort_rev for descending order.

```
array      sort(list);
table      sort(<table>,<column>);
```

UNARY OPERATORS

Unary operators operate on one value. The precedence of a unary operator is less than the DOT or CALL operators. The unary operators consist of + - * / and CAST.

The CAST operator converts, as best it can, from one type to another. Casting a string to Varbinary means to copy the string without the null, for the strings length. Converting a numeric to a string will use the BEST format to do the conversion. A conversion from datetime to double will translate the time from CAS time to SAS time. Converting anytime related value to double and back will preserve the correct time as either CAS time (integer) or SAS time (double).

BINARY OPERATORS

Binary operators are the heart of the expression parser. The parser supports the standard set of operations, including operations found in DATA step and operations to allow subsetting of arrays, dictionaries and result tables. The precedence is used to control the order of operations. Please see the CASL manual to see all available operators. These operators are polymorphic. The data type of the operands to an operator controls what operation is performed. CASL defines parenthesis, brackets, and braces as operators. The binary operators use is defined below:

1) Parentheses are used to change the order of operations. Use these when you are unsure of the precedence of an operation.
2) The left and right brackets are used for array and directory indexing. They are used to evaluate the index values for an array or dictionary reference. Each dimension is separated by a comma.
3) The left and right braces are used to create arrays and dictionaries.

DATA TYPE AND THEIR EFFECT ON THE EXPRESSION PARSER

Let's look at how data types affect the operation of the expression parser.

The plus operator:

- A number plus a number is addition.
- A string plus a string is string concatenation.
- An array plus a number concatenates the number onto the end of the array.
- A dictionary plus a dictionary will concatenate the two together removing any duplicates from the second dictionary.

The division operator:

- Division of numeric values results in division.
- Division of two arrays results in adding the values in the 2nd array that do not exist in the 1st array.

The DOT (.) operator is the workhouse of the expression parser. The two operands define the operations performed. Note that while parsing, left and right brackets are converted to DOT operators. This means indexing of arrays and dictionary lookup uses the DOT operator. Let's look at the operations that help with processing results:

- dictionary.name results in a dictionary lookup. The dictionary.name is replaced by the value defined in the dictionary.
- dictionary.func(a,b) expects 'func' to be a function variable. The function in that variable is called.
- array[number] results in array index lookup.
- Result_table.name uses the result table's dictionary to result the lookup.
- Result_table[number] extracts that row as a dictionary.
- Result_table[,"name"] extracts that column as an array.
- Result_table.where(<expr>) applies the given expression to each row of the result table. Each time the expression is true the new result table will keep that row.
- Result_table.compute("column name", "column type", "column format", <expr>) applies the given expression to each row of the result table. The result of the expression is added as a new column to the new result table.

Sometimes the two operands do not go together such as a result table divided by 2. In these cases, the result will be a missing value. If a function is not defined, then the expression will fail. It was decided that failure was a better solution than not calling the function.

An expression can be very complicated, but evaluation is very simple. Keep in mind the simple evaluation rules. If you forget the precedence, use parentheses to force the precedence you want applied to the expression evaluation.

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 the same syntax as represented in SAS

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

If a variable is assigned to a format, that variable can now 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 value is 1.23
```

You can modify the format of any column by using the 'format' statement.

```
  format tab MSRP best8. HP cylinders best12.;
```

Now the format for MSRP is BEST8. and for HP and CYLINDERS is BEST12. The format statement might also be used to set the default format for a variable.

LET'S PRODUCE A REPORT

It has been a long journey to understand the knowledge needed to process CAS action results. You should understand:

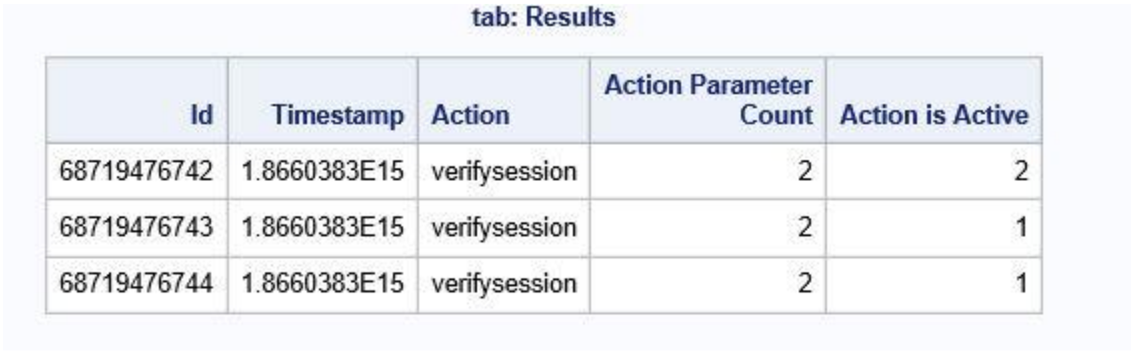
- What is a result?
- What are the data types that make up the result?
- What operators can be used in the expression parser to process results?
- What are some of the functions you can use to process results?
- How formats are used to change the way a variable is displayed.

Our 1st example will look at translating valid from a result table into something understandable. The next example uses the cars data set to produce a finished report.

Let's start easy. I have an action that returns a result table the has the following fields:

```
 1)  ID                  [Id                ]       (int64)
 2)  timestamp           [Timestamp ]               (int64)
 3)  action              [Action Name        ]      (varchar)
 4)  parameterCount      [Action Parameter Count ] (int64)
 5)  active              [Name is Active]           (int64)
```

This action describes the details of a list of actions. I need to produce a report that has more content. The active and ID fields have information that needs to be transformed. The active field is either 0,1 or 2 to signify disconnected, active, running. The ID is really two 32-bit fields in one 64-bit integer. I need to split them up to display their real value. To do this I will to add 3 new fields and remove two fields. How can CASL help me achieve this goal? Here is the output:


tab: Results

Id	Timestamp	Action	Action Parameter Count	Action is Active
68719476742	1.8660383E15	verifysession	2	2
68719476743	1.8660383E15	verifysession	2	1
68719476744	1.8660383E15	verifysession	2	1

If my result table is in the variable 'tab', let's examine the code to produce a new result table that has the information I want to display.

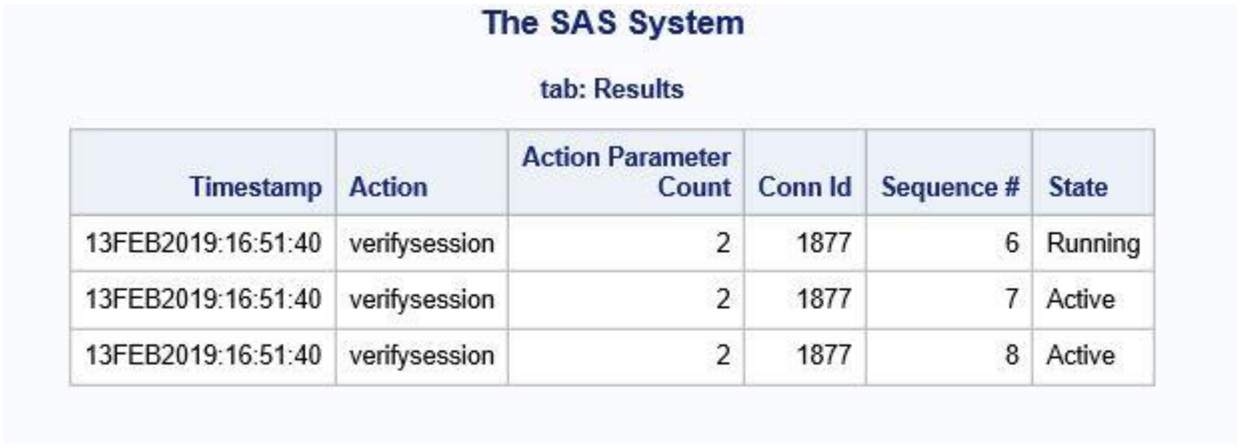
```
format tab Timestamp DATETIME19.;
function determinestate( active);
if (active == 1) then return("Active");
if (active == 0) then return("Disconnected");
if (active == 2) then return("Running");
return("Unknown");
end;

tab = tab.compute({"conn","Conn Id",best.},  (int64)(id / (2**32)));
tab = tab.compute({"seq","Sequence #",best.},  id - conn*(2**32))[;
tab = tab.compute({"state","State",$10.}, determinestate(active));

tab = tab[ ,(tabcolumns(tab)-{"ID","active"})];

print tab;
```

The refined table results are:



The SAS System

tab: Results

Timestamp	Action	Action Parameter Count	Conn Id	Sequence #	State
13FEB2019:16:51:40	verifysession	2	1877	6	Running
13FEB2019:16:51:40	verifysession	2	1877	7	Active
13FEB2019:16:51:40	verifysession	2	1877	8	Active

This refined report has the columns I wanted, with the formats I wanted and the values are informative.

LET EXPLORE WHAT THIS CODE DOES TO PRODUCE THE RESULTS

The first line sets a format for the 'timestamp' field. It is a datetime, but I want it displayed as DATETIME19. I then define a function that translates a 0,1,2 into a string. This will translate the 'active' field into a string that represents the state of the action.

The next operations use the 'compute' operator to create three new columns in the table.

1) A connection id, with name 'conn'. This is the 1st 32 bits of the ID value.
2) A sequence number, with the name 'seq'. This is the 2nd 32 bits of the ID value.
3) Translate the active value to a string, with the name 'state'.

Once I added the new columns, I need to remove the 'active' and 'ID' fields. I subset the result table reducing the columns with an expression that gets the list of column names as array and subtract the ID and active fields from that list. Note that I used parenthesis around the expression to get the precedence correct for the subtraction. The operations here were simple, but they allowed me to take a result table that had obscure data and transform it into a finished result. You need to determine where the data is that you want and how do you want to present it.

HOW TO CREATE YOUR OWN RESEULT TABLE FROM SCRATCH

PROC CAS supports a function to create a new result table. This is a powerful way to create your own results tables from scratch. You can use the 'tabcolumns' and 'tabtypes' functions to extract the variable names, label names, and column data types from a table. This allows you to save those values as arguments to the 'newtable' to duplicate the structure of any table.

You need to supply:

1) Name of the table
2) A list of the names for the variables.
3) A list of data types for each variable.
4) You might then supply rows to be added.

```
 t=newtable( "table", { "make", "MSRP", "HP", "cylinders" },
                  { "varchar", "int64", "integer", "int64"},
                  {"dodge", 20000, 250, 4},
                  {"ford  ", 30000, 200, 6} );
```

At this point you might want to see if the format of the result table is correct. This is where you can use the DESCRIBE statement to display the format of a given expression.

```
describe t;run;
NOTE: The result table 'table' has been created.
[table] Table ( [2] Rows [4] columns
Column Names:
[1] make         [make            ] (varchar)
[2] MSRP         [MSRP            ] (int64)
[3] HP           [HP              ] (int32)
[4] cylinders    [cylinders       ] (int64)
```

You can add a new row by executing the following code:

```
          addrow(t,{“honda”,19000,197, 4});
```

PRINTING AND ARCHIVING A RESULT TABLE

Once you have a result table the contains the report data you want, what can you do with this result table. The print statement allows you to print the result. The saveresult statement allows you to save result tables and varbinary data in many forms. This allows you to read that result back into CASL at a later date. Here are your options for printing and archiving a result table.

1) Print the result table using ODS. ODS is the interface used in SAS® Studio to display result tables. ODS understands how to use the attributes in a result table to display the results as any SAS® Procedure would display them. This is the primary method of printing a result table.

   ```
   print results.fetch;
   ```

2) Print the result table to the log bypassing ODS. You can instruct CASL to print the table in a simple 4 column per row format to validate that the data you expect to be in the result table is there. ODS uses a template to format the data, which might hide columns depending on the template. The ‘output log’ statement forces the results to be printed to the log. This can be canceled by using the ‘output ods’ statement.

```
output log;
print results.fetch;
```

3) Save the result table as a CSV file. CASL will create a separate file for varbinary rows and columns in the table. These files will end with the .row#_col# extensions. The columns for these varbinary will be changed to the path of the file for each column and row.

```
saveresult tab csv="./table.csv";
```

4) Save the result table as a SAS data set, resulting in a .sas7bdat file. The varchar columns will be converted to CHAR columns. This data set can now be used as a normal SAS data set.

```
saveresult tab dataout=mytable;
saveresult tab lib=mylib;
```

5) Save the result table as a binary file. The result table is saved as a binary file. The read_pathname function can be used to read the result table back into CASL.

```
saveresult tab bin file=”table.rt”;
tab = read_pathname(“table.rt”,”table”);
```

6) Save the result table as a text file. All the fields will be converted to a string in the text file. This allows you to save a copy of the result table as a printed version of the table. The default is not to print the label of the columns. This allows a user to print a result table, which contains one column of DATA step code to a file and then execute that DATA step.

```
filename filer “file.txt”;
Saveresult tab file=filer;
Saveresult tab file=”data.sas”;
```

7) Upload the result table to the CAS server as a CAS table. The result table can be uploaded into the CAS server. You specify the name of the resulting CAS table and the caslib where should be placed. This is useful if you intend to create or modify an existing result table and need to run an action on the modified table.

```
Saveresult tab cas=”casout” caslib=”casuser”;
```

CONCLUSION

In conclusion, you have seen how CASL language provides a good environment to both submit actions to the CAS server and process the results into finished reports. The choice of a language to access the CAS server should be based on how you intend to use that language to achieve the results you want. CASL was designed from the start to process CAS actions. The operators and functions provide a wide range of options to help produce reports the show the results you need to show. CASL running PROC CAS offers you the option to run your jobs in SAS studio and have access to all of the features of Base SAS. The development of CASL will continue to add new features specifically to help get your job done.

REFERENCES

.

CASL Programmers Guide:
https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=proccas&docsetTarget=titlepage.htm&locale=en

PROC CAS and CASL Reference Documentation:
https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=proccas&docsetTarget=titlepage.htm&locale=en

SAS®

CONTACT INFORMATION

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

Jerry Pendergrass
Analytic Server Research and Development
Distinguished Software Developer
919-531-7766
Jerry.Pendergrass@sas.com