Paper 081-2025

Beyond Macro - Data-Driven Programming with CAS in SAS® Viya®

Mark Jordan, SAS Jedi (Retired)

ABSTRACT

With the adoption of SAS Viya, accelerating processing in CAS is becoming more common, and CAS speaks CASL. Seasoned SAS coders often use SAS macros to produce data-driven programs, automating tedious programming tasks. When working with CAS from the SAS Compute Server, it's important to know where, when, and how the CASL that executes is generated. Whatever your experience level, the interactions between SAS code, CASL, and Macro can be intimidating. This presentation aims to demystify that process.

INTRODUCTION

In this paper I'll discuss:

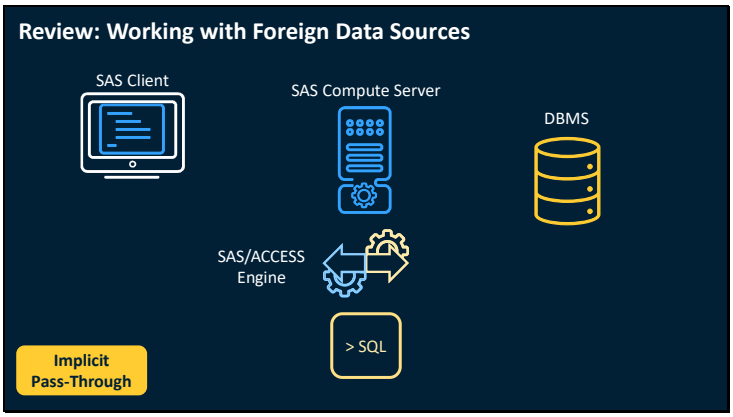
- A brief review of code tokenization and macro triggers on the SAS Compute Server.
- A brief introduction to CASL and CAS actions, including the effect of macro triggers in CAS programs.
- An introduction to CASL variables and data types, including arrays and dictionaries.
- Practical techniques for advanced data-driven programming in CASL

DATA-DRIVEN PROGRAMMING

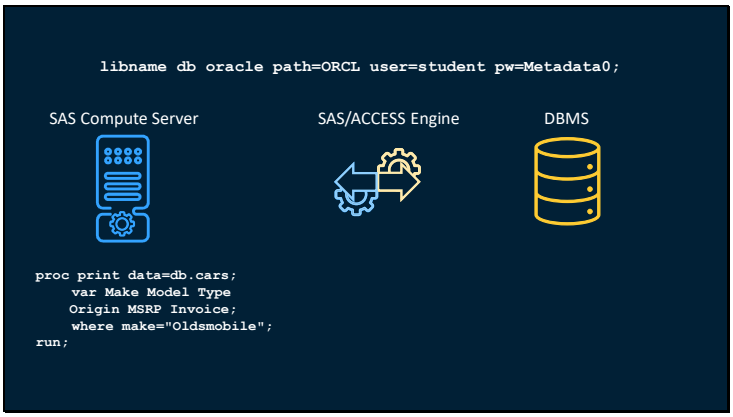
For SAS programmers, processing in CAS is becoming much more common, and CAS speaks a different native language called "CASL". Seasoned SAS coders like me love to use SAS macro to generate data-driven code and to automate otherwise tedious programming tasks. It's important that we know where and how our CASL code is being generated. No matter what your level of experience, the interactions between SAS code, CASL, and Macro can sometimes be intimidating. So, let's peek behind the curtain to get a better feel for how, where, and when these pieces are all connected.

Beyond Macro - Data-Driven Programming with CAS in SAS® Viya®

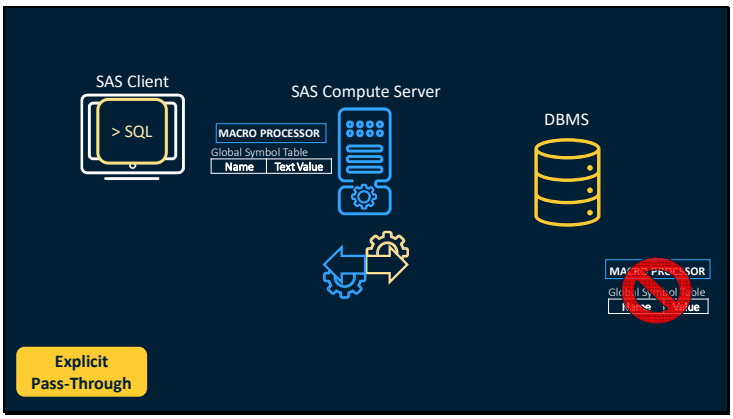
Mark Jordan, SAS Jedi (Retired)



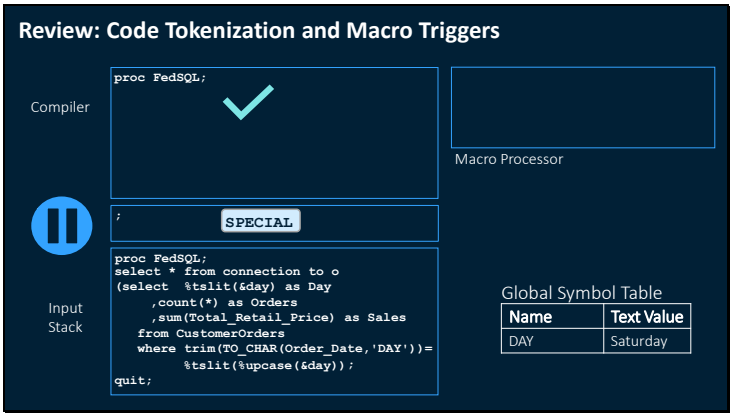
Before diving into CASL, I want to quickly review some SAS/Access and macro concepts, because there are some similarities in how these things work. For example, when working with foreign data sources, like database tables, the SAS/ACCESS LIBNAME engine provides transparent access via implicit pass-through. We write and submit our SAS code to the SAS Compute server. There, the LIBNAME engine translates as much code as possible into native database SQL, which is passed to the database for execution.



In this example, the libref DB provides transparent access to an Oracle database. We write and submit a PROC PRINT step on the SAS Compute server. The LIBNAME engine translates the code into native database SQL and passes it to the database for execution.

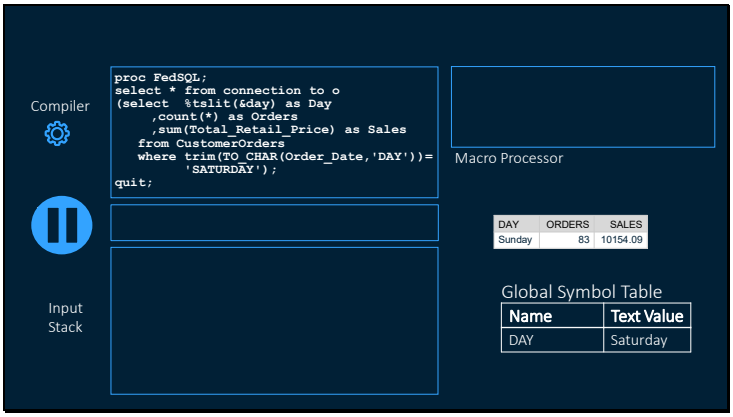


We can also choose to write the database-specific SQL ourselves using PROC SQL or PROC FedSQL. The database SQL is not syntax checked by the SAS Compiler. Instead, the database SQL code is forwarded as text to the database, where the SQL is interpreted and executed by the database system. But what if pass-through SQL contains macro triggers? The SAS macro facility is built into the SAS Compute server. It isn't available to the database.

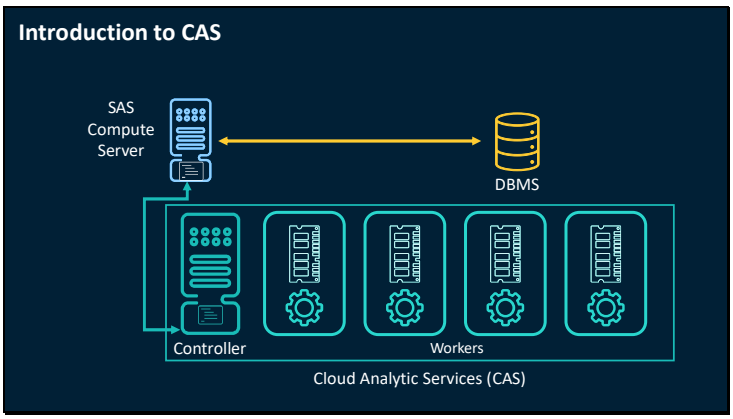


So, if the database has no macro processor, how does macro code in the pass-through SQL get processed? This pass-through query contains some macro code. Before it can successfully turn the code, we provide as text into machine code, the compiler needs a little bit of metadata added to determine how to handle the process. In SAS, the word scanner parses your code and appends a little bit of metadata to each chunk in a process called tokenization. It then sends the tokens to the compiler for processing. The word scanner grabs the text up to the first delimiter, analyzes the content, adds the appropriate metadata, and send the token to the compiler. The word scanner then grabs the next bit of text up to the next delimiter and repeats the process sending the new token to the compiler. This process continues until the compiler sees that it has a complete statement. At this point, tokenization pauses as the compiler checks the statement syntax, and if no syntax errors are detected, the process resumes.

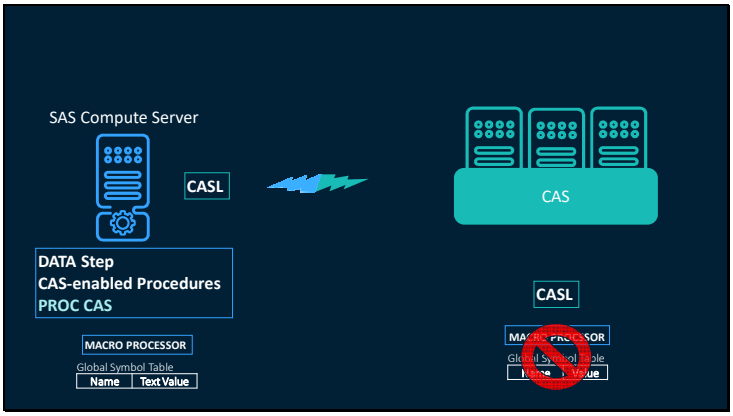
If the word scanner encounters macro triggers, communication with the compiler is paused, and tokens are sent to the macro processor instead. The macro processor resolves macro variables and executes macro code to generate text which is pushed back onto the top of the input stack.



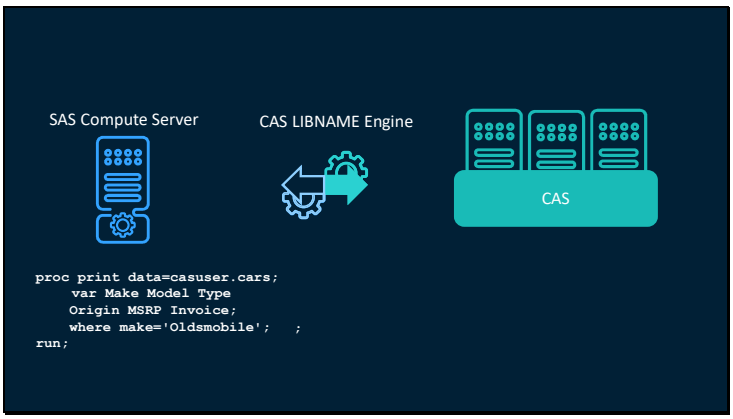
Tokenization resumes until the compiler encounters a run boundary. Then tokenization pauses and the compiler executes the program step. The pass-through SQL is sent directly to the database as text. The SAS compiler does not check syntax of this code. When the database finishes processing, the result set is returned to SAS for further processing. So, as you can see, all macro code and variable references were resolved by the Compute Server's macro processor *before* the pass-through SQL code was sent to the database. The database never sees the macro code.



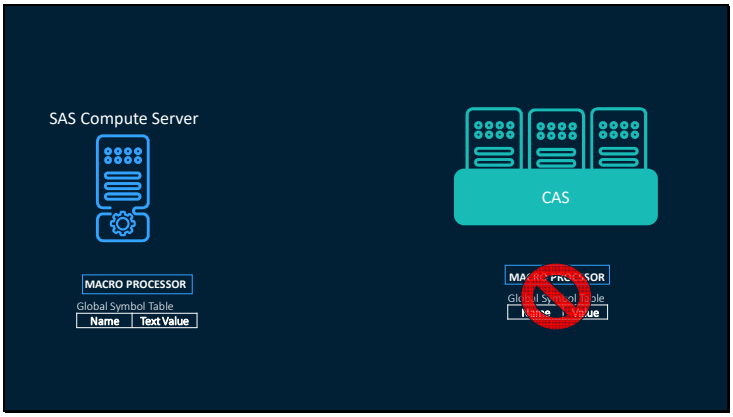
In Viya, we also have access to Cloud Analytic Services, or CAS for short, which is Viya's native, massively parallel processing engine. CAS tables are pre-loaded into memory and pre-distributed across the CAS workers. Properly written programs on the Compute Server can generate code that is executed in CAS.



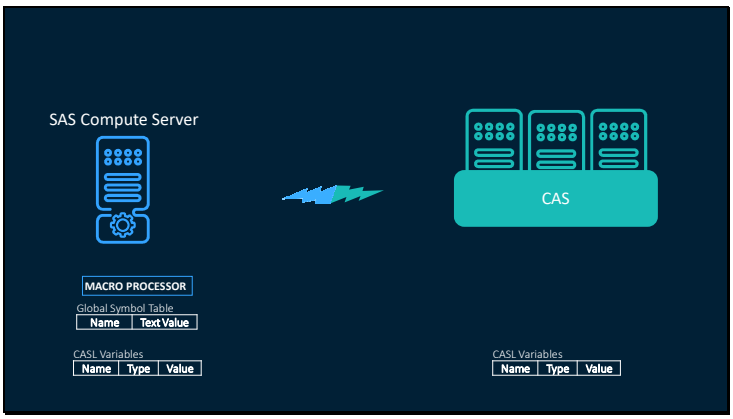
Unlike most databases, the native language of CAS is not SQL. Instead, CAS "speaks" CASL. There is a CAS engine available on the Compute Server, and in a process analogous to using a database engine, your Compute Server DATA step and CAS-enabled procedure code can generate CASL on your behalf. Alternatively, you may choose to explicitly write your own CASL code using PROC CAS. As you probably surmised, the code you submit to the Compute Server still gets parsed by the Word Scanner, so the macro facility still functions normally, and macro processing is carried out on the Compute Server before the CASL code is sent to CAS. The SAS macro facility is not available on the CAS server itself.



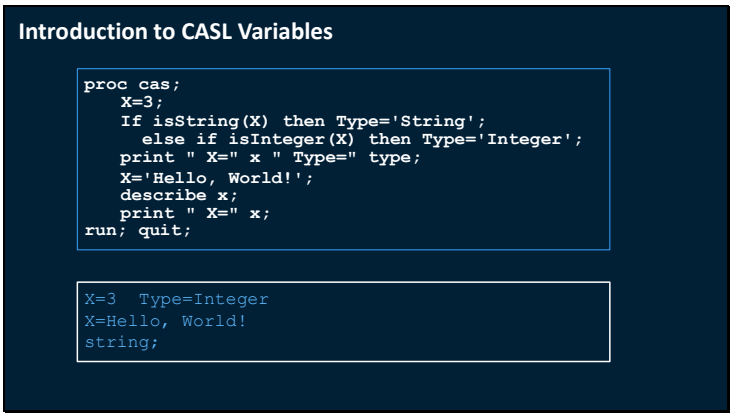
When working with CAS-enable processes and CAS tables on the SAS Compute server, the CAS LIBNAME engine provides transparent access via implicit pass-through. We write and submit our SAS code to the SAS Compute server. The LIBNAME engine translates as much code as possible into native CASL, which is then passed to the CAS server for execution.



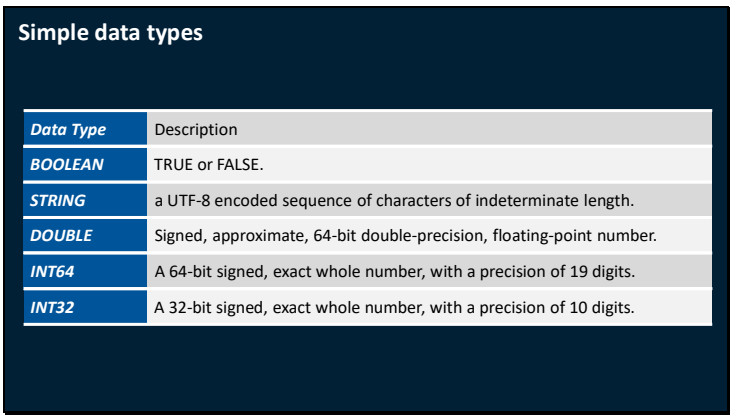
It's important to remember that the macro facility lives only on the Compute Server. CAS has no macro facility.



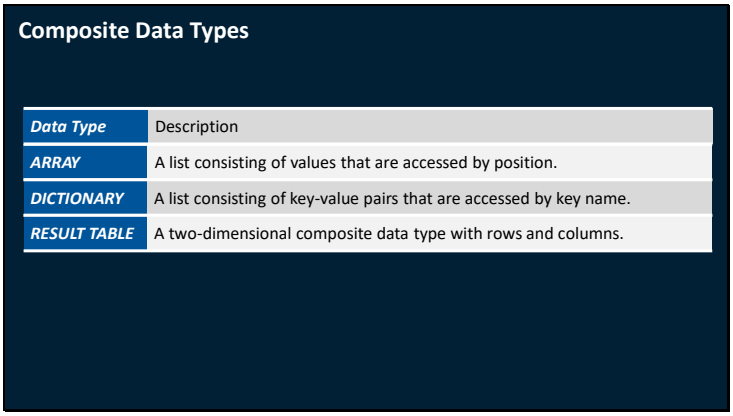
When programming with PROC CAS on the Compute Server, using macro variables and macro definitions is one way to generalize or generate CASL code before sending it to CAS. In that case, things will work pretty much as you expect. The CASL language provides some macro-style functions but also includes a robust CASL Variable infrastructure that provides a significantly more flexible and robust data-driven programming capability. And here's where it starts to get a little tricky. PROC CAS can execute some CASL statements on the Compute Server without requiring an established CAS connection, while some CASL, including all CAS actions, can only be execute on the CAS server. And it's difficult to distinguish where your CASL code ran by merely looking at the SAS log. CASL that executes on the Compute Server can create CASL variables in Compute Server memory, and CAS actions often return their results to the Compute Server as CASL variables.



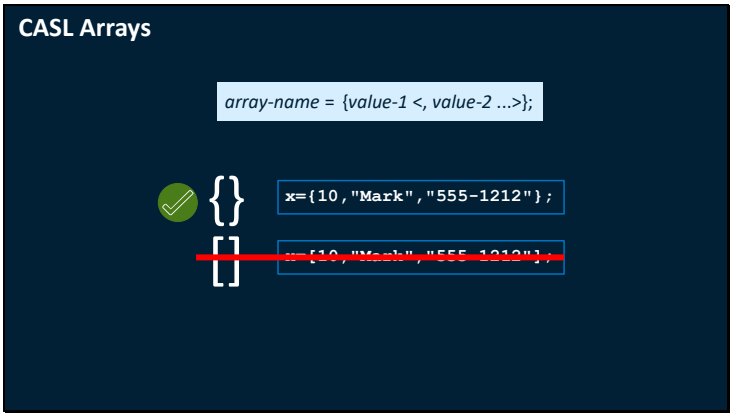
CASL uses dynamically typed variables, with the interpreter assigning a variable's type at runtime based on the value you assign. You can modify a variable's type on the fly merely by assigning another value of a different type. In this example, we assign the numeric value 3 to variable X. We then use CASL isType functions and a PRINT statement to examine the variable's data type and value. Next, we assign the character value "Hello, World!" to variable X and use a PRINT statement to examine the variable's value and a DESCRIBE statement to report the variable's data type. The log reveals that the type of X was set to numeric by the first value assignment and dynamically changed to string when a text value was subsequently assigned.



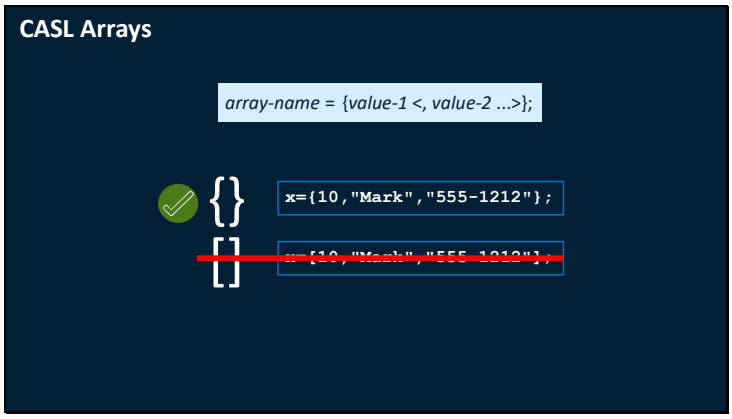
CASL provides a variety of data types useful for application development. Boolean variables hold one of two values: TRUE or FALSE. Variables with text values assigned are typed as string. These variables contain text of indeterminate length. Numeric variables with fractional values are stored as double. By default, numeric values without a fraction are stored as INT64, but you can declare variables designed to hold smaller values as INT32 instead.



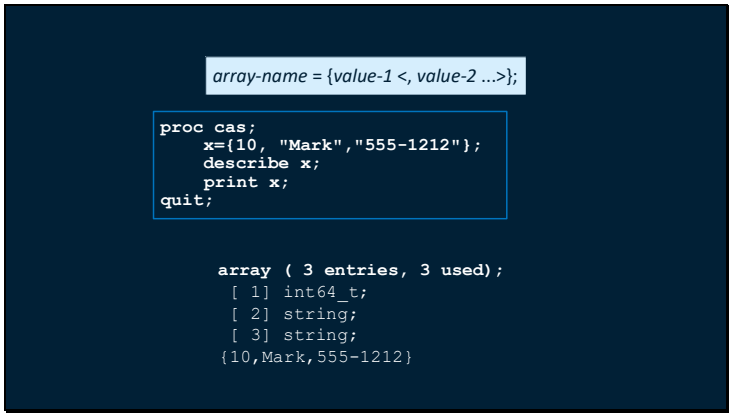
But the real power and usefulness of CASL variables rests in composite data types. An array is an ordered list of values, with individual elements accessed by position. A dictionary is an unordered list of key-value pairs, with individual elements accessed by key value. And finally, result tables are a two-dimensional data type consisting of rows and columns, with column names acting as keys. Table rows are accessed by position and table columns by key value, so it's useful to conceptually picture a result table as an array of dictionaries.



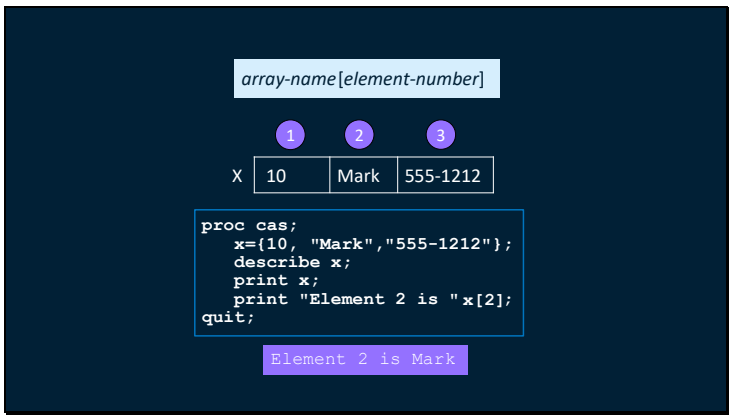
A CASL array is an ordered list of values. You can use an assignment statement to create an array in CASL by specifying the array name to the left of the equal sign and a comma-separated value list to the right. Individual values in an array are referred to as elements or items. Seasoned SAS programmers will notice differences between CASL and DATA step arrays. In some languages, you can use square brackets or curly braces to define an array, but CASL accepts only curly braces for CASL array definitions. In some languages, all elements of an array must have the same data type, but this is not true of CASL arrays.



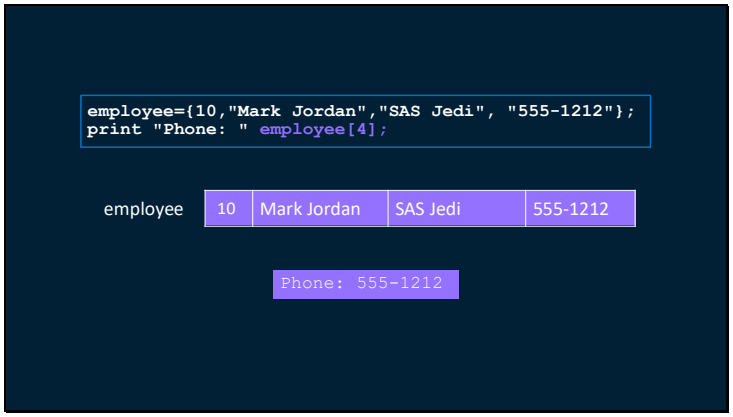
A CASL array is an ordered list of values. You can use an assignment statement to create an array in CASL by specifying the array name to the left of the equal sign and a comma-separated value list to the right. Individual values in an array are referred to as elements or items. Seasoned SAS programmers will notice differences between CASL and DATA step arrays. In some languages, you can use square brackets or curly braces to define an array, but CASL accepts only curly braces for CASL array definitions. In some languages, all elements of an array must have the same data type, but this is not true of CASL arrays.



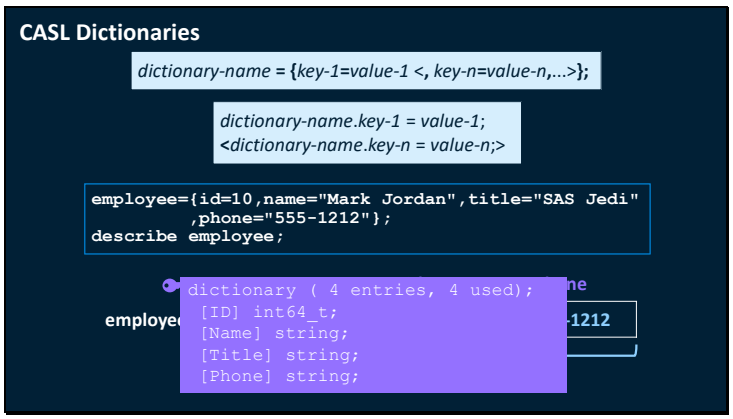
Here we create an array named x and specify three values, 10, Mark, and 555-1212. Because the last two values are strings, they're enclosed in quotation marks. Next, we use the DESCRIBE and PRINT statements to view the data type and values of x. In the log, the DESCRIBE statement output shows that the variable x is an array with three entries. The first element is INT64, and the remaining two elements are strings. The PRINT statement wrote the values of the array to the log: 10, Mark, and 555-1212, surrounded by curly braces, indicating that this is an array.



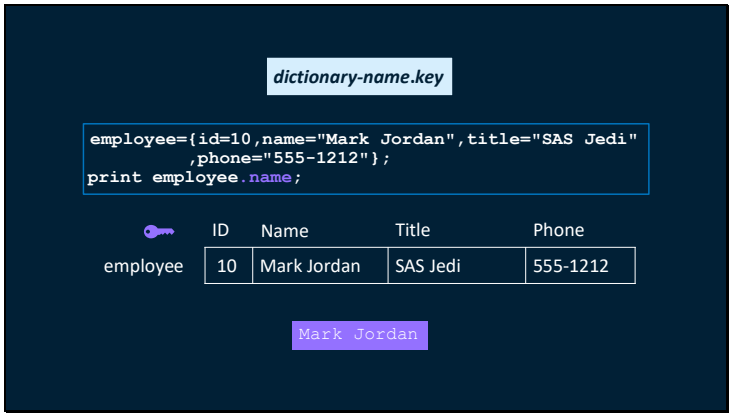
After an array is created, you can access its elements in code. To access the value of an array element, we can specify the array name with the element number in square brackets. CASL array element numbering begins at 1, not zero. In the example code, the reference x[2] will yield the value "Mark".



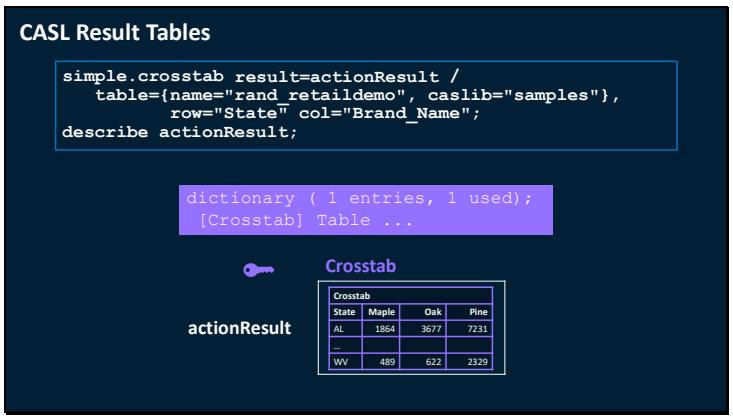
This is an array named employee. In has a numeric element followed by three string elements. You can probably figure out that the first string is a name, maybe the second is a job title, and the third a phone number. To access the phone number, you could use employee[4]. Accessing elements by position is OK for small arrays, but what if the array had twenty elements? And how can you easily discern the significance of the values stored there? For example, the purpose of the first numeric value in this array is not clear. This would be a lot easier if the elements just had names...



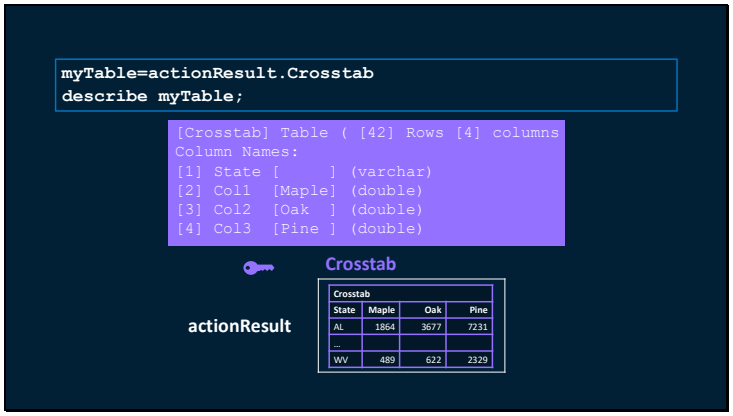
A CASL dictionary is an unordered list of key-value pairs. Values can be retrieved by using the key as a name. The easiest method to define a CASL dictionary is to use a comma-separated list of key-value pairs surrounded by curly braces. You can also use dot notation, but this requires a separate assignment statement for each key. This example creates a dictionary named employee, with 4 keys: ID, Name, Title, and Phone and an appropriate value assigned to each key.



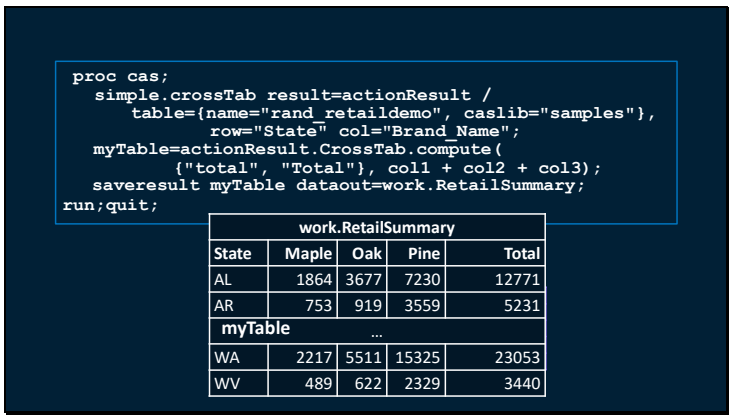
Individual items are usually retrieved using dot notation consisting of the dictionary name, a dot, and the key value. This code creates a dictionary named **employee** and assigns values to keys named **ID**, **Name**, **Title**, and **Phone**. The subsequent print statement retrieves the value from the **employee** dictionary' key named **name** and prints the value in the log.



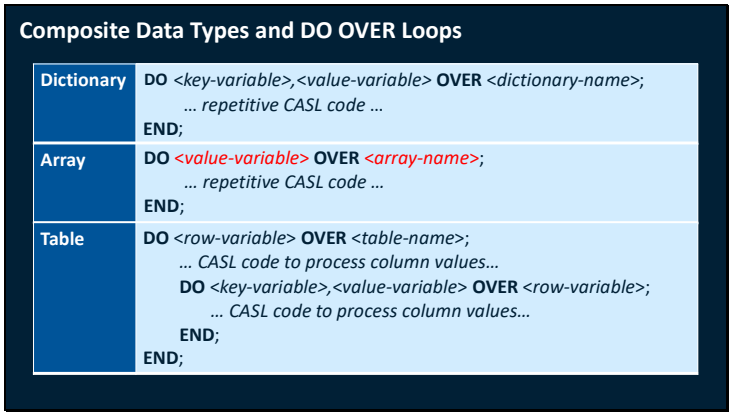
Although it is possible to create, read, and manipulate them with code, CASL result tables are most often encountered in the results of a CAS action. CAS action results are generally returned in a dictionary, and one or more of the dictionary keys reference a result table.



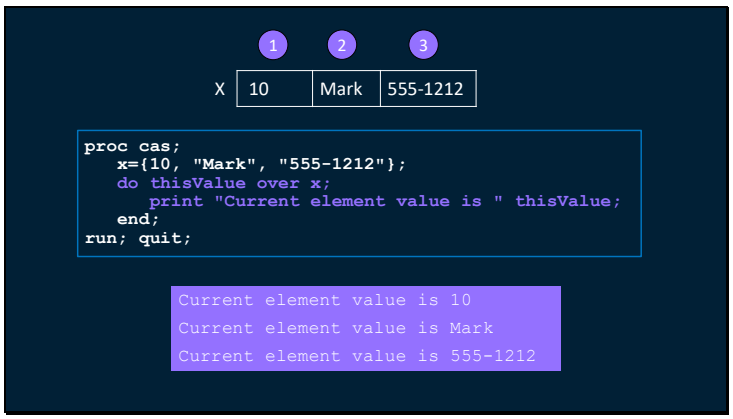
If we extract the table to a variable, in this example, myTable, we can continue working with just the result table.



A nice feature of working with CAS actions using the Compute Server as a client is that, while the actions are all processed in CAS, the results are returned as CASL variables to the Compute Server and can be further manipulated there. Here, we add a computed Total column to the result table, then save the result table directly to a SAS data set named work.RetailSummary.



In CASL, you can use an iterative DO OVER loop to step through the elements in a CASL composite data type. The DO OVER loop starts with the keyword DO. If stepping through a dictionary, next is the name of the key-variable that will receive the name of the current key, and then the name of the value-variable which will receive the value associated with the key. These variables will be assigned to a new value during each iteration. Next is the required keyword OVER followed by the dictionary-name, specifying the name of the dictionary to be processed. You can place as many CASL statements and actions as you like before the required END statement, which indicates the end of the loop. The process described between the DO OVER and END statements will execute repetitively, once for each element in the dictionary. An array has no keys, so to process an array, just leave off the key-variable and specify an array-name instead of a dictionary name. And because a table can be conceptualized as an array of dictionaries, you can use the array technique to access rows, and the dictionary technique to work through each row's key-value pairs.



Let's look at a simple example. This DO OVER loop iterates over the elements of array X, using a value-variable named thisValue. Inside the loop, we print the current value of thisValue to the Log. On the first iteration, thisValue is assigned the value of the first array element, the PRINT statement writes the value 10 to the log, and execution returns to the top of the loop. On the next iteration, thisValue is assigned the value of the second array element, the PRINT statement writes the value "Mark" to the log, and execution once again returns to the top of the loop. Next iteration, thisValue is assigned the value of the last array element, the PRINT statement writes "555-1212" to the log, and the loop stops executing.

Let have a look at using Macro and CASL variables to create data-driven, dynamic code in SAS Viya.

Note:

All SAS code, Results (html), and Log (txt) files are available in the downloadable ZIP file for this presentations.

DEMO 1 – CAS-ENABLED PROCS GENERATE CASL

Code:

```
%let myMake=Oldsmobile;

/* From a SAS Data Set */
title "PROC PRINT - sashelp.cars";
proc print data=sashelp.cars;
   var Make Model Type Origin MSRP Invoice;
   where make="&myMake";
run;

/*****************************************************************************
 What if the data was in a database instead of a SAS Data Set?
*****************************************************************************/
/* SASTRACE allows us to see what SAS actually "said" (SQL) to the database */
options sastrace=',,,d' sastraceloc=saslog nostsuffix;

/* Using implicit pass-through to PostgreSQL - SAS?ACCESS writes the SQL */
title "PROC PRINT - db.cars";
proc print data=db.cars;
   var Make Model Type Origin MSRP Invoice;
   where make="&myMake";
```

```
run;

/* Using explicit pass-through to PostgreSQL - write your own SQL */
title "PROC SQL with Explicit pass-through to PostgreSQL";
proc SQL;
connect using db;
select *
   from connection to db
   (select Make, Model, Type, Origin, MSRP, Invoice
      from CARS
      where make=%tslit(&myMake));
disconnect from db;
quit;
options sastrace=off;

/*****************************************************************************
 Now, what if the data was in a CAS table instead?
*****************************************************************************/
/* From a CAS table */
title "PROC PRINT from CAS";
proc print data=casuser.cars;
   var Make Model Type Origin MSRP Invoice;
   where make="&myMake";
run;
/* See what SAS actually "said" to CAS */
cas casauto listhistory 2;

/* Let's do this explicitly using CASL and CAS Actions */
proc cas;
/* Copy the CAS Action code from the log */
title "PROC CAS - table.fetch CAS action";
table.fetch /
   table={name='CARS', caslib='casuser', where="(make='Oldsmobile')"
        ,vars={{name='Make'}
            ,{name='Model'}
             ,{name='Type'}
            ,{name='Origin'}
            ,{name='MSRP'}
            ,{name='Invoice'}}}
      ,from=1
       ,to=428
      ,sasTypes=false
      ,index=false
;
quit;

/* There aren't really 428 rows in the output. Can we be more precise? */
proc cas;
/* Create a CASL variable with the table specs for ease of use */
myTable={name='CARS', caslib='casuser', where="(make='Oldsmobile')"
        ,vars={{name='Make'}
            ,{name='Model'}
             ,{name='Type'}
            ,{name='Origin'}
            ,{name='MSRP'}
            ,{name='Invoice'}}};
```

```
simple.numrows result=count
   /table=myTable;
print count;
print "NOTE: Simple.numrows produced a dictionary named COUNT with one entry named
NUMROWS.";
print "The value of count.numrows is " count.numrows;
quit;

proc cas;
title "PROC CAS - table.fetch CAS action with data-driven TO parameter";
/* Create a CASL variable with the table specs for ease of use */
myTable={name='CARS', caslib='casuser', where="(make='Oldsmobile')"
        ,vars={{name='Make'}
            ,{name='Model'}
             ,{name='Type'}
            ,{name='Origin'}
            ,{name='MSRP'}
            ,{name='Invoice'}}};

simple.numrows result=count/
   table=myTable;

table.fetch /
   table=myTable
         from=1,
      /* Use count.numrows here to insert the correct value */
       to=count.numrows
      ,sasTypes=false
      ,index=false
;
run; quit;
cas casauto listhistory 2;
title;
```

Notes:

All of the methods produce the same results. When we run PROC PRINT on the database table, the SAS/ACCESS engine generated PostgreSQL SQL code to produce the results. When we ran PROC PRINT against the CAS table, the CAS engine generated CASL with CAS actions used to produce the results. Using LISTHISTORY, we can see and copy the CASL from the SAS log and write the CASL code ourselves. This allows us to customize the results and, when submitted, runs a bit more efficiently because the CAS engine is no longer required to generate the code.

DEMO 2 – DATA-DRIVEN CASL

Code:

```
/*************************************************************************
 Data-driven CASL
*************************************************************************/
/************************************************************************
 Section 1.a: Load all files from a caslib's data source using SAS Macro
*************************************************************************/
/* To start, make sure all CAS tables are unloaded */
```

```
proc datasets library=casuser kill nolist nodetails;
run; quit;

/* Traditional macro techniques */
/* Open macro definition loadfiles.sas and review, then run this section */

%include "&path/macros/loadfiles.sas" /source2;

options mprint;
%loadFiles(casuser,&path/data)

title "Tables in CASUSER";
proc casutil;
   list tables incaslib="casuser";
run; quit;


/*************************************************************************
 Section 1.b: Load all files from a caslib's data source using CASL &
              CAS actions.
*************************************************************************/
/* Clear the decks for the next section */
proc datasets library=casuser kill nolist nodetails;
run; quit;
/* Show that CASUSER has no tables loaded in the Libraries pane */
proc cas;
   title "Files available to be loaded in CASUSER";
   table.fileInfo/caslib="casuser";
   title;
run; quit;

/* CASL result tables and a DO OVER loop */
proc cas;
thisCaslib="casuser";
/* List files in the caslib, result in a dictionary */
table.fileInfo result=rFileInfo/caslib=thisCaslib;

/* Extract table of filenames (named FileInfo) from the result dictionary */
files=rFileInfo.FileInfo;

/* Loop over dictionary, load the files as CAS tables */
do key, value over files;
   print "Loading " value.name " to CASUSER";
   table.loadTable/
      caslib=thisCaslib
      path=value.name
      casout={name=scan(value.name,1,'.'),caslib=thisCaslib,replace=TRUE}
   ;
end;
title "Tables loaded in CASUSER";
table.tableInfo/
   caslib=thisCaslib;
title;
run;
/*************************************************************************
 Section 2.a: Print a few rows of every table in a CASLIB using SAS Macro
*************************************************************************/
```

```
%macro listCASTables(CASlibref);
proc sql noprint;
select distinct strip(memname)
   into :table1-
    from dictionary.tables
   where libname="%qupcase(%superq(CASlibref))"
;
quit;

proc cas;
/* Generate a table.fetch call for 5 rows of each table */
%do I=1 %to &SQLObs;
table.fetch/
   table={caslib="%superq(CASlibref)", name="&&table&i"}
   ,to=5
   ,index=FALSE
;
%end;
run; quit;
%mend;

/*Call the macro for the CASUSER libref */
%listCASTables(casuser);

/*************************************************************************
 Section 2.b: Print a few rows of every table in a CASLIB using CASL &
              CAS actions.
*************************************************************************/
/* CASL - no macro required */
proc cas;
table.tableInfo result=resultDictionary/
   caslib="casuser";

/* Iterate over the result table without first extracting it from the result
dictionary */
do thisTable over getcolumn(resultDictionary.TableInfo, "Name");
   table.fetch/
      table={caslib="casuser", name=thisTable}
      ,to=5
      ,index=FALSE
;
end;
run; quit;
title;
```

Notes:

The CASL code required for this job is more succinct than the macro code and requires fewer steps. Because it doesn't need the CAS engine to generate CASL, it will also run faster. But the macro is nice because, after it is created, I can use it like a macro function and the code required is just a one-liner. Can I make the CASL into a user-defined function?

DEMO 3 – DATA-DRIVEN CASL USER-DEFINED FUNCTIONS

Code:

```
/*****************************************************************************
 Data-driven CASL User-defined Functions
*****************************************************************************/
/*****************************************************************************
 Section 1: Roll your own CASL function.
*****************************************************************************/
proc cas;
function printTables(caslib);
   table.tableInfo result=rDict/caslib=caslib;
      do thisTable over getcolumn(rDict.TableInfo, "Name");
         table.fetch/
             table={caslib=caslib, name=thisTable}
            ,to=5
            ,index=FALSE
         ;
      end;
end;

/* Verify that CAS knows about my function */
functionlist printTables;

/* Use my new function to print tables in the SESUG caslib */
printTables("casuser");
run; quit;

/*****************************************************************************
 Section 2: Functions don't persist after the PROC CAS session quits. Use
 readpath() and execute() to recreate all functions from a single code file.
*****************************************************************************/
/* Unload all of the tables in CASUSER */
proc datasets library=casuser kill nolist nodetails;
run; quit;
```

Notes:

When using PROC CAS from the SAS Compute server, CAS variables and user-defined functions persist only until the procedure QUIT statement. To create user-defined functions that can be used across multiple sessions, we first create and save a file containing all of the function code:

Code for the user-defined function file:

```
function printTables(thisCaslib);
   table.tableInfo result=rDict/caslib=thisCaslib;
   do thisTable over getcolumn(rDict.TableInfo, "Name");
      table.fetch/
          table={caslib=thisCaslib, name=thisTable}
         ,to=5
         ,index=FALSE;
   end;
end;
function loadAllFiles(thisCaslib);
   /* List files in the caslib, result in a dictionary */
   table.fileInfo result=rFileInfo/caslib=thisCaslib;
```

```
   /* Extract table of filenames (named FileInfo) from the result dictionary */
   files=rFileInfo.FileInfo;

   /* Loop over dictionary, load the files as CAS tables */
   do key, value over files;
   print "Loading " value.name " to " thisCaslib;
   table.loadTable/
      caslib=thisCaslib
      path=value.name
      casout={name=scan(value.name,1,'.'),caslib=thisCaslib,replace=TRUE}
      ;
   end;
   table.tableInfo/caslib=thisCaslib;
end;
```

Notes:

Then, using the READPATH and EXECUTE, we can easily re-create all of the functions, then use them just like native CASL functions in our code.

Code:

```
proc cas;
   /* Read in & execute the User Defined Functions code */
   myUDF = readpath("&path/udf/casl-udf.sas");
   execute(myUDF);
   /* Use the functions as if they were built into CAS */
   loadAllFiles("casuser");
   printTables("casuser");
run; quit;
```

CONTACT INFORMATION

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

Mark Jordan
Email: sas.jedi@gmail.com
X: @SASJedi