SAS® Cloud Analytic Services 3.1: CAS Procedure Programming Guide and Reference

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Cloud Analytic Services 3.1: CAS Procedure Programming Guide and Reference*. Cary, NC: SAS Institute Inc.

SAS® Cloud Analytic Services 3.1: CAS Procedure Programming Guide and Reference

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2016

3.1-P1:proccas

Contents

Chapter 1

CAS Procedure

Overview: CAS Procedure

About the CAS Procedure

The CAS procedure enables you to interact with SAS Cloud Analytic Services (CAS) from the SAS Client by providing you programming environment based on the CASL language specification. The programming environment enables you to run CAS actions and use the results to prepare the parameters for another action.

The CAS procedure has several features that enable you to perform the following operations:

- Run any CAS action that is supported by the server, even if the action did not exist at the time of the release.

  Note: upload and addTable actions are currently not supported.
- Use multiple sessions to perform asynchronous execution.
- Operate on parameters and results as variables using the full function expression parser.

About the CAS Language

- CASL is the language specification that enables you to access the CAS server. CASL is an integral part of the CAS procedure. It is designed to look and feel like the SAS language, such as DATA step, IML, and CMP.
- The language is case insensitive when referencing variables, either created by CASL or as a result of an action. You cannot define the data type for a variable in a variable declaration. The data type of a variable is determined by the type of value that is assigned to it.
- The values that are in CASL are CAS values that enable you to pass the values as parameters to the CAS server. CASL also supports stand-alone arrays and dictionaries.

Terminology

action
: A task that is performed by the server at the request of the user. An action sends a request to the server, which parses the arguments of the request and invokes the action function. The result is returned and resources are cleaned.

action set
: A collection of actions (tasks) that group functionality such as session management, table management, and so on.

argument
: A value that is supplied to a procedure when it is called to perform an operation.

condition
: One or more numeric or character expressions that result in a value on which some decision depends.

expression
: An expression is used in program statements to create, assign, calculate, and transform new values and to perform conditional processing.

function
: Is a group of statements that perform a task together. A function takes a variable and a parameter and returns a value.

results table
: A table is created as the result of an action, and can be used for other actions. In addition to rows and columns, the results table also contains labels, attributes, and variable types.

CAS session
: A single period during which a software application is in use, from the time that the application is invoked until its execution is terminated.

statement
: A type of SAS language element that is used to perform a particular operation in a SAS program or to provide information to a SAS program.

variable
: A symbolic name for a value. The value can be a list, a dictionary, or a simple data types (string, integer, or floating-point number). You can assign a value throughout a program.

Syntax: CAS Procedure

```
PROC CAS;
   ACTION <action-set-name.action-
     name><RESULT=<variable>><STATUS=<rc>><ASYNC=name>
     / <parameters>;
   ASSIGNMENT target = expression;
   CALL function (argument1, argument2...);
   DESCRIBE variable;
   DO;
      ... more CASL statements ...;
      END;
   DO UNTIL
      condition;
      ... more CASL statements ...;
      END;
   DO WHILE condition;
      ... more CASL statements ...;
      END;
   DO [<key,>] <var> OVER <value>;
```

```
        ... more CASL statements ...;
        END;
    DO <var> = <start> [TO <stop>] [BY <increment>]
        [WHILE <condition> | UNTIL <condition>];
        ... more CASL statements ...;
        END;
    FILE;
    FUNCTION function name ( [argument 1 [, argument 2...] ] );
        ... more CASL statements ...;
        END;
    GOTO label;
    IF expression
        THEN statement;
        ELSE < statement>;
    LOADACTIONSET actionsetname;
    ON condition response expression;
    OUTPUT libname;
    PRINT expression;
    RAISE condition-name;
    RETURN <expression>;
    SAVERESULT variable-name <noreplace> <dataout=<libref.> member-name> |
      <lib=libref> |
      <file = <path-name| filename >|<caslib=<caslib>>|<casout=name>;
    SELECT select-expression; <when-list> ... <when-list>; OTHERWISE statement-
      list; END end-label; <when-list> :: = WHEN <when-expression> <statement-list>;
    SESSION session-name;
    SOURCE variable; text; endsource;
UNSET <DISP> <LOGS>;
RUN;
QUIT;
```

Statement	Task
PROC CAS	Enables you to interact with the CAS server from SAS using the CAS language and CAS actions.
ACTION	Runs SAS Cloud Analytic Services actions.
ASSIGNMENT	Assigns a value to a variable.
CALL	Calls function with the specified arguments.
DESCRIBE	Displays the data type of a variable created by the ACTION statement.
DO	Creates a block of code that executes as one statement.
DO UNTIL	Executes statements in a DO loop repetitively until a condition is true.

Statement	Task
DO WHILE	Executes statements in a DO loop repetitively until a condition is true.
DO OVER	Iterates over a list or result table.
DO	Executes statements in a DO loop repetitively until a condition is true.
END	Ends a DO group or SELECT group processing.
FILE	Specifies different locations of output.
FUNCTION	Creates a new function that can be called in an expression.
GOTO	Directs program execution immediately to the statement label that is specified.
IF-THEN/ELSE	Executes a SAS statement for observations that meet specific conditions.
LOADACTIONSET	Loads a SAS Cloud Analytic Services action set. Some actions are available as platform-level functionality. You can use the LOADACTIONSET statement to load more action sets into your session.
ON	Enables you to specify how to handle an error during execution.
OUTPUT	Writes the current observation to a SAS data set.
PRINT	Displays the value of the expression to the current output location.
RAISE	Sends a signal to the calling process or thread.
RETURN	Returns a value from the current function.
SAVERESULT	Creates a SAS data set from the results of an ACTION.
SELECT	Executes one of several statements or groups of statements.
SESSION	Specifies an existing session to use for running a program.
SOURCE	Embeds text in the program and assigns the program to the given variable.
UNSET	Controls message suppression in the SAS log.

PROC CAS Statement

Enables you to program and schedule SAS Cloud Analytic Services actions from the SAS client.

Syntax

PROC CAS;

ACTION Statement

Runs SAS Cloud Analytic Services actions.

Note: You can use expressions to build up parameters for ACTION statements. To use an expression in a statement, wrap the expression in a dollar sign ($), followed by a parentheses ( ).

See: For documentation about the actions that you can run with the ACTION statement, see: *SAS Cloud Analytic Services: System Programming Guide*, *SAS Cloud Analytic Services: Analytics Programming Guide*, *SAS Cloud Analytic Services: Data Mining and Machine Learning Programming Guide*, *SAS Cloud Analytic Services: Statistics Programming Guide*

Example: "Example 2: Run an Action" on page 30

Syntax

<ACTION> *<action-set-name.>action-name* <RESULT= *<variable>* <STATUS = *<rc>*
<ASYNC = *name>*
/ *<parameters>*;

Required Argument

<action-set-name.> action-name
specifies the action to run.

action-set-name
specifies the name of the action set that contains the action to run. Examples are **`tables`**, **`simple`**, and **`network`**.

action-name
specifies the name of the action to run. Examples are **`loadTable`**, **`listActions`**, and **`serverStatus`**.

Example For example, the following statement specifies the loadtable action that is part of the tables action set:

```
action tables.loadtable / path="cholesterol.csv";
run;
```

In this example, PATH= is a parameter of the loadtable action.

Optional Arguments

parameter
specifies the specified action's parameters. You can list the parameters for an action with the help action.

Example

```
action help submit / action="action-name";
run;
```

RESULT=*variable-name*
stores the results of the action in a variable. You can then use the variable in other CASL statements and actions.

Alias R=

See For examples of using results in a program, see "Working with Results and Creating Variables" in *SAS Cloud Analytic Services: System Programming Guide*.

STATUS= <*variable-name*>
stores the status code of the action in a variable. You can use the variable with statements such as DESCRIBE and with a selected set of SAS statements within the scope of the CAS procedure invocation. If you do not specify a *variable-name*, a variable named **`_status`** that contains the status codes is created.

See For information about status codes, see "Severity and Reason Codes" in *SAS Cloud Analytic Services: System Programming Guide*.

ASYNC=*name*
enables you to submit multiple requests to the server. If your requests are in the same session, the requests are queued in the order received. If your requests are in separate sessions, the requests are executed in parallel.

ASSIGNMENT Statement

Evaluates an expression and stores the result in a variable.

Tip: If the variable already exists, the new assignment replaces the variable and the old value is overwritten.

Syntax

<ASSIGNMENT> *variable* = *expression*;

Actions

variable
names a new or existing variable.

expression
specifies the expression used in program statements to create variables, assign values, calculate new values, transform values, and perform conditional processing.

Tip expressions can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign.

Details

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

Example: Valid Assignment Statements

This assignment statement defines the target variable as “numeric”.

```
x = "numeric";
```

This assignment statement defines the target variable as a numeric value of 88.

```
xx.y = 88;
```

This target statement defines the target variable as *z*.

```
z = min(y, 70);
```

CALL Statement

Calls a function with the specified argument. If the function returns a value, it is ignored.

Syntax

<CALL> *function argument-1* <*argument-2, …*>;

Optional Arguments

function
processes one or more arguments and can be used in either an assignment statement or an expression.

arguments
specifies a value that is supplied to a procedure when it is called to perform an operation.

DESCRIBE Statement

Displays the contents of a variable created by the ACTION statement.

Syntax

DESCRIBE *variable*;

Required Argument

variable
specifies the name of a variable created by the ACTION statement.

See “ACTION Statement” on page 6

Examples

Example 1: Using the DESCRIBE Statement

The following example stores the result of the serverStatus action in the variable named *r*.

```
action serverstatus result=r submit;
run;
describe r;
run;
```

The DESCRIBE statement displays the result as a list that includes three entries: about (a list), server (a table), and nodestatus (a table).

```
List ( 3 entries, 3 used);
[About] List ( 6 entries, 6 used);
[CAS] String;
[Version] String;
[Built] String;
[Copyright] String;
[System] List ( 6 entries, 6 used);
[Hostname] String;
[OS Name] String;
[OS Family] String;
[OS Version] String;
[OS Release] String;
[Model Number] String;
[Documentation] String;
[server] Table ( [1] Rows [2] columns
Column Names:
[1] Node Count (Double)
[2] Total Actions (Double)
[nodestatus] Table ( [143] Rows [5] columns
Column Names:
[1] Node Name (String)
[2] Role (String)
[3] Uptime (Sec) (Double)
[4] Running (Double)
[5] Stalled (Double)
```

You can access the variable *r* and learn about the server.

Example 2: Producing Results Using Previously Assigned Variables

In the previous example, you stored the results in the variable *r* and received three entries. You can take one of the entries and print the data.

```
print r.server [1];
run;
```

The output gives you information about the CAS server.

```
{nodes=143 , actions=17}
```

DO Statement

Creates a block of code that executes as one statement.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Example: "Example 3: Define a New Function" on page 31

Syntax

DO;

... more CASL statements ...;

END;

Details

The DO statement is the simplest form of DO group processing. The statements between DO and END statements are called a DO group. You can nest DO statements within DO groups. The statements between the DO and END statements are called a DO group. A simple DO statement is often used within an IF-THEN/ELSE statement. See "IF-THEN/ELSE Statement" on page 16. A DO statement is executed depending on whether the IF condition is true or false.

Example: Using the DO Statement

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 7. If YEARS is less than or equal to 7, statements in the DO group do not execute, and the program continues with the assignment statement that follows the ELSE statement.

```
if years>7 then
   do;
      months=years*12;
      put years= months=;
   end;
   else do;
yrsleft=7-years;
end;
```

DO Statement, Iterative Statement

Iterates over a list.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Tips: The order of the optional TO and BY clauses can be reversed.

When you use more than one specification, each one is evaluated before its execution.

Example: "Example 6: Computing Partial Dependency" on page 36

Syntax

DO <*variable*> = <*start*> TO<*stop*>
<BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>;

… more CASL statements …;

END;

Optional Arguments

variable
names a variable whose value governs execution of the DO group.

start
specifies the initial value of the variable.

When both start and stop are present, execution continues (based on the value of increment) until the value of index-variable passes the value of stop. When only start and increment are present, execution continues (based on the value of increment) until a statement directs execution out of the loop, or until a WHILE or UNTIL expression that is specified in the DO statement is satisfied. If neither stop nor increment is specified, the group executes according to the value of start. The value of stop is evaluated before the first execution of the loop.

stop
specifies the ending value of the index variable.

BY *increment*
specifies a positive or negative number (or an expression that yields a number) to control the incrementing of index-variable.

The value of increment is evaluated before the execution of the loop. Any changes to the increment that are made within the DO group do not affect the number of iterations. If no increment is specified, the index variable is increased by 1. When increment is positive, start must be the lower bound and stop, if present, must be the upper bound for the loop. If increment is negative, start must be the upper bound and stop, if present, must be the lower bound for the loop

WHILE (*condition*) | UNTIL(*condition*)
evaluates, either before or after execution of the DO group, any condition that you specify. Enclose the condition in parentheses.

A WHILE expression is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. An UNTIL expression is evaluated after each execution of the loop, so that the statements inside the group are executed repetitively until the expression is true.

Restriction A WHILE or UNTIL specification affects only the last item in the clause in which it is located.

Details

There are four other forms of the DO statement:

- The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The DO OVER statement iterates over a list.
- The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.
- The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.

Example: Using the Do Loop Statement

This statement identifies the loop variable as **i**.

```
do i=1 to 10;
```

This statement executes the loop variable in increments of 3.

```
do i=1 to 10 by 3;
do i=1 to 10 while( i < 5);
do i=1 to 10 by 1 until ( i>=5);
```

DO OVER Statement

Iterates over a list.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Example: "Example 4: Use Run-time Math Functions" on page 32

Syntax

DO <key>, <var> OVER <value>;

... more CASL statements ...;

END;

Optional Arguments

key

Used to place the key or index for the list.

var

Used to access the variable contexts.

value

Used to reference the list.

Example: Using the DO OVER Statement

The following example takes the results of an action, iterates over the results to find Nmiss values, and then saves the values in a list. That list can then be used as input to another action. For the complete example, see "Example" in *SAS Cloud Analytic Services: System Programming Guide*.

```
action simple.summary result=CPSSum                    /*1*//
      table={caslib="casdata", name="phoneSubs"};
run;


cinfo = findTable(CPSSum);                             /*2*/
   nmissVar = {"Country Name", "Indicator Name"};      /*3*/

   do col over cInfo;                                  /*4*/
      if (col.NMiss <20) then do;                      /*5*/
          nmissVar= nmissVar + col.Column;             /*6*/
      end;
   end;
   print nmissVar;                                     /*7*/
run;
```

1. The Summary action creates descriptive statistics such as sum, means, and nmiss. The RESULT= option saves the results in a variable named CPSSum.
2. Cinfo is the name of the variable that contains the results of the FINDTABLE function. FINDTABLE is an internal function that searches for a value in a table. CPSSum is the variable that holds the results of the Summary action.
3. NmissVar is the name of the variable that will hold the results of the DO OVER statement. The brackets { } indicate that a list is being made. Country Name and Indicator Name are columns to be included in the list. The rest of the columns are added by the DO OVER statement.
4. The DO OVER statement iterates over the results stored in the variable cInfo. Col is the name of the index for the list.
5. The IF THEN DO statement finds columns that have a value of NMiss greater than twenty.
6. The columns are then added to the list in the variable nmissVar.
7. The PRINT statement prints the variable nmissVar to the SAS log.

Output 1.1 *SAS Log*

```
{Country Name,Indicator Name,1990,1995,1996,1997,1998,1999,2000,
2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014}
```

DO UNTIL Statement

Executes statements in a DO loop repetitively until a condition is true.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Syntax

DO UNTIL *condition*;

... any CASL statements ...;

END;

Required Argument

condition

specifies one or more numeric or character expressions that result in a value on which some decision depends.

Example: Using a DO UNTIL Statement to Repeat a Loop

This statement repeats the loop until **x** is greater 10. The expression *x>10* is evaluated at the bottom of the loop. There are ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

```
x=0;
do until (x>10);
print x;
x = x+1;
```

```
end;
```

DO WHILE Statement

Executes statements in a DO loop repetitively while a condition is true.

Restriction: An END statement must follow a DO statement and all of the DO group processing statements.

See: "DO Statement, Iterative Statement" on page 10

Syntax

DO WHILE condition;

... any CASL statements ...;

END;

Required Argument

condition

One or more numeric or character expressions that result in a value on which some decision depends.

Example: Using a DO WHILE Statement

These statements repeat the loop while **x** is greater than 5. The expression *x > 10* is evaluated at the top of the loop. There are ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

```
x=0;
do while (x > 10);
print x;
x = x + 1;
end;
```

END Statement

Ends a DO group, FUNCTION, or SELECT group processing.

Syntax

END;

Without Arguments

Use the END statement to end DO group, FUNCTION, or SELECT group processing.

Details

The END statement must be the last statement in a DO group, FUNCTION, or a SELECT group.

FILE Statement

Enables you to specify different locations in the output.

Example: "Example 5: Develop a Scorecard Using Actions " on page 33

Syntax

FILE ;

Required Argument

FILE
Bypass ODS for CAS result tables and send output to the desired location.

Examples

Example 1

```
file tablefile "u/user/table.lst";

print table;
```

Example 2

Sets the active output location to ODS.

```
file ods;
```

Example 3

Sets the active output location to the file.

```
file nodes "/u/user/nodes";

action listnodes result=r submit;
```

The result of the listnodes action is printed to **`/u/user/nodes`**.

```
print r;
```

Example 4

Set the active output location to the SAS log.

```
file log;
```

The same results are printed to the log.

```
print r;
```

FUNCTION Statement

Creates a new function that can be called in an expression.

Requirement: An END statement must follow a FUNCTION statement.

Example: "Example 3: Define a New Function" on page 31

Syntax

FUNCTION *function- name*(*argument–1* , *argument–2*);

statements;

END;

Required Arguments

function- name
specify a user-defined function name.

argument
specifies the value that is supplied to a procedure when it is called to perform an operation.

statement
is any CASL statement.

GOTO Statement

Directs program execution to the specified statement label.

Syntax

GOTO *label*;

Required Argument

label
specifies a statement label that identifies the GO TO destination.

IF-THEN/ELSE Statement

Executes a SAS statement for observations that meet specific conditions.

Example: "Example 3: Define a New Function" on page 31

Syntax

IF *expression* **THEN** *statement*;

<ELSE ><*statement*>;

Required Arguments

expression
Used in program statements to create variables, assign variables, calculate new values, transform variables, and perform conditional processing.

statement
Any CASL statement.

Details

CAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing. A nonzero and nonmissing result causes the expression to be true; a result of zero or missing causes the expression to be false.

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a statement for observations that are read from a table. An optional ELSE statement gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement. Using IF-THEN statements without the ELSE statement causes CAS to evaluate all IF-THEN statements.

Using IF-THEN statements with the ELSE statement causes CAS to execute IF-THEN statements until it encounters the first true statement. Subsequent IF-THEN statements are not evaluated.

Note: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

Example

```
x=5
if (x > 6) then put "x is greater than 6";
else print "x is less than or equal to six";
print "x=" x;
run;
```

LOADACTIONSET Statement

Loads a SAS Cloud Analytic Services action set. Some actions are available as platform-level functionality. You can use the LOADACTIONSET statement to load more action sets into your session.

Requirement: The ability to load action sets into your session is subject to access controls.

Example: "Example 5: Develop a Scorecard Using Actions " on page 33

Syntax

LOADACTIONSET "*action-set-name*";

Required Argument

action-set-name
specifies the name of a SAS Cloud Analytic Services action set.

See For documentation about the actions that you can add with the LOADACTION statement, see: *SAS Cloud Analytic Services: Data Mining and Machine Learning Programming Guide* and *SAS Cloud Analytic Services: Statistics Programming Guide*.

ON Statement

Enables you to specify error handling during execution.

Syntax

ON *condition response expression*;

Required Arguments

condition response
An expression that evaluates an error and produces a value.

expression
An expression is used in program statements to create variables, to assign values, to calculate new values, to transform variables, and to perform conditional processing.

Optional Arguments

The supported conditions are as follows:

unknown
This condition is raised after accessing a variable with an unknown value.

illop
This condition is raised after performing an invalid operation.

Example

The following example demonstrates registering a function named Gety. In the function, the ON statement is used to register a response for handling unknown variables—in this case, to replace an unknown value with a replacement value, 3.2. Because the z variable is referenced, but undefined, the response handler for unknown values is triggered.

```
proc cas;
repval = 3.2;
function gety(x) do;
on unknown value=repval response=replace;
y=z;
return (y);
end func;
x = 1;
print "y value = " get y(x);
run;
```

The SAS log includes the following message:

```
y value = 3.2
```

OUTPUT Statement

Sets the active output location.

Example: "Example 6: Computing Partial Dependency" on page 36

Syntax

OUTPUT libname;

Required Argument

libname
specifies the name of the location to which SAS writes the observation. The available choices for locations are ODS (default) and the log.

PRINT Statement

Displays the value of the expression to the current output location.

Example: "Example 4: Use Run-time Math Functions" on page 32

Syntax

PRINT <*expression*> <*expression*>;

Optional Argument

expression
An expression with no more than one *operator*. An expression can consist of one of the following single operators: constant, variable, function. You can include more than one expression in your PRINT statement.

Example

This example prints the variable r. The variable has to be previously defined.

```
print r;
```

RAISE Statement

Sends a signal to the calling process.

Note: Raise ( ) returns a 0 on success and a nonzero on failure.

Syntax

RAISE *condition*;

Required Argument

condition
One or more numeric or character expressions that result in a value on which some decision depends.

RETURN Statement

Returns a value from the current function.

Example: "Example 5: Develop a Scorecard Using Actions " on page 33

Syntax

RETURN <expression>;

Optional Argument

expression
Used in program statements to create variables, assign variables, calculate new values, transform variables, and perform conditional processing.

Details

The RETURN statement can be used in conjunction with the "IF-THEN/ELSE Statement" on page 16 and the "FUNCTION Statement" on page 15 .

SAVERESULT Statement

Creates a SAS data set from the results of an *ACTION*.

Note: A column that contains more than 32,000 characters is too large to save in a data set.

Example: "Example 5: Develop a Scorecard Using Actions " on page 33

Syntax

SAVERESULT *variable-name* <NOREPLACE>
<DATAOUT=<*libref.*> *data-set-name*> | <LIB=*libref*> |
<FILE=<*path-name* | *filename*>| CASLIB=<*caslib*>| CASOUT=*name*>;

Required Argument

variable-name
includes one table result or a list of table results

Optional Arguments

NOREPLACE
specifies not to overwrite saved files.

DATAOUT=<*libref.*>*data-set-name*
specifies the SAS library and data set name.

LIB=*libref*
specifies which SAS library to use.

FILE=<*path-name* | *filename*>
specifies the file or filename to use.

CALIB=<*casref*>
specifies which caslib to use.

CASOUT=*name*
specifies the name of the result table.

SELECT Statement

Executes one of several statements or groups of statements.

Restriction: An END statement must follow a SELECT statement.

Syntax

SELECT < *select-expression*)> ;

<when-list> ... *<when-list>* ;

OTHERWISE*statement-list* ;

END *end-label* ;

<when-list> :: =

WHEN *<when-expression>* *<statement-list>* ;

Required Arguments

select-expression
specifies an expression that evaluates to a single value.

end-label
the END statement closes the SELECT statement. The optional end-label argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the SELECT statement, or an error will occur.

statement-list
can be any executable SAS statement, including DO, SELECT, and null statements. You must specify the *statement* argument.

when-expression
specifies any SAS expression, including a compound expression.

Note: You must specify at least one *when-expression*.

when-list
can be any SAS compound expression containing at least one *when-expression*.

Details

Using WHEN Statements in a SELECT Group

The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify CAS statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Note: SELECT statements can be nested.

Example: SELECT with a select-expression

This example illustrates how to use the SELECT statement with a *select-expression*.

```
select (a);
when (1) x=x*6;
when (2);
when (3,4,5) x=x*20;
otherwise;
end;
```

SESSION Statement

Specifies an existing session to use for running a program.

Example: "Example 5: Develop a Scorecard Using Actions " on page 33

Syntax

SESSION *session-name*;

Required Argument

session-name
specifies the name of an existing session.

Examples

Example 1: Print Session ID

This example prints your session ID for the *casauto* session.

```
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
cas casauto;
proc cas;
    session casauto;
    action sessionid result=uuid;
    print uuid;
run;
```

```
{TESTSESS:Tue Sep 13 11:16:18 2016=4ebd26ad-9dc4-cf4f-a108-4f497d06a23f}
```

Example 2: List Session Options

This example lists all the options for the *casauto* session.

```
cas casauto LISTSESSOPTS;
```

Example 3: Connect to an Existing Session

```
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
cas casauto uuid="4ebd26ad-9dc4-cf4f-a108-4f497d06a23f";
proc cas;
    session casauto;
    session.sessionstatus result=s;
```

```
    put s;
run;
```

SOURCE Statement

Enables you to embed text in the program and assign it to a given variable.

Syntax

SOURCE *variable*;

<text>;

endsource;

Required Argument

variable
A symbolic name for a value. The value can be a list, a dictionary, or a simple data types (string, integer, or floating-point number). You can assign a value and you can assign new values throughout a program.

Optional Argument

text
String value from the variable.

UNSET Statement

Controls message suppression in the SAS log.

Syntax

UNSET <DISP> <LOGS>;

Optional Arguments

DISP
specifies to suppress printing the disposition in the SAS log.

LOGS
specifies to suppress printing messages from CAS in the SAS log.

Details

You can specify **`UNSET DISP LOGS`** to suppress both types of messages.

Example

The following SAS log shows how **UNSET DISP** is used to suppress the disposition from the SAS log:

```
60 proc cas;
61 session mysess;
62
63 setsessopt / caslib="foo";
64 run;
NOTE: Active Session now mysess.
WARNING: The value foo for casLib is invalid.
ERROR: The CAS server stopped processing this action because of errors.
Disposition: Severity Error
65 /* suppress the "Disposition:" message in the SAS log */
66 unset disp;
67 setsessopt / caslib="foo";
68 run;
WARNING: The value foo for casLib is invalid.
ERROR: The CAS server stopped processing this action because of errors.
```

Using: CAS Procedure

Arrays

An array is a list of values. An array is created when you assign a value to one of the elements in the array. The number of items in the array is set by the highest index. All indexes in the array start at 1 and all the values do not have a key. Values that are not set are *uninitialized*. Elements in an array are referenced with brackets [ ].

There are no limits to the number of dimensions that you can have for an array. For example, a two-dimensional array is a list of lists.

This is an example of an array:

```
sample[4,5,6] = 14
```

This example creates a three-dimensional array. The third dimension has the highest index of 4.

```
arrayex = x[2,3,4]
```

This is an example of an uninitialized value. The array index can be a normal expression within CASL. This example would result in an error.

```
box[3*4+5, box, function(4,5,6)] = 15;
```

Note: Numeric values, including constants, are always double unless explicitly created as int64 values.

Dictionary

A dictionary is a list of values that have a key name with a value. Dictionaries can be multidimensional.

Here are the two acceptable syntaxes for dictionaries:

```
a.foo.bar=1;

a["foo","bar"= 1];
```

Expression Parser

Expression Evaluation

The CASL language supports a full featured expression evaluation system. Each variable is a location to hold values and type information goes with the values. The result of an expression evaluation is a value. The value can be assigned to a variable. These are the rules for expression evaluation:

- **`x = 5 + 5`**; is an integer.
- **`x = 5 + 5.0`**; is a double.
- **`x = “5” + 5`**; is a double.
- **`x = (int64) “5” + 5`**; is an integer.
- **`6/2`**; is a double.

Note: Numeric values, including constants, are always doubles unless explicitly created as int64 values.

Variable and Literal Mode

By default the expression parser treats all names as variable names. A value must be enclosed in quotation marks to be a literal string of the name.

```
model = "MODEL";
make="MAKE";
mpg="MPG";
```

Functions

Overview

The CASL language supports function calls within expression evaluation. A function can take multiple arguments. The function can return an error if the correct number of arguments were not provided. Variables that were created during the execution of a CASL function are local to that function. The CASL interface provides you with run time support to create and manage values that the routine may return. Functions can be defined in many ways:

- CASL supports internal functions that provides run-time support for your CASL program.
- The FUNCTION statement (FNC) can be used to defined functions using the CASL syntax that accepts a parameter and returns a value.
- PROC CAS may install functions to support native operations.
- CASL provides a default set of functions for regular operations. You can replace these functions with your own.
- DS2 functions are supported and you can use the FNC statement to get a list and description of the functions.

FNC Statement

The FNC statement is used to list all available functions by name and category with simple descriptions.

The following code displays all available functions within the FNC statement.

```
1? proc cas;
2? fnc cate;
3? run;
```

This is the output of all available functions.

```
array
bitwise
char
combinatorial
datetime
distance
financial
math
probability
quantile
random
special
statistics
trig
truncation
```

```
4? fnc bitewise; run;
```

```
band          (bitwise)    Returns the bitwise logical AND of two arguments.
blshift       (bitwise)    Returns the bitwise logical left shift of two
arguments.
bnot          (bitwise)    Returns the bitwise logical NOT of an argument.
bor           (bitwise)    Returns the bitwise logical OR of an argument.
brshift       (bitwise)    Returns the bitwise logical right shift of two
arguments.
bxor          (bitwise)    Returns the bitwise logical EXCLUSIVE OR of two
arguments.
```

```
5? fnc bnot;
6? run;
```

```
bnot          (bitwise)    Returns the bitwise logical NOT of an argument.
    double  bnot(double);
```

```
8? fnc "interest";
9? run;
```

```
compound      (financial)  Returns compound interest parameters.
comipmt       (financial)  Returns the cumulative interest paid on a loan
between the
                           start and end period.
effrate       (financial)  Returns the effective annual interest rate.
ipmt          (financial)  Returns the interest payment for a given period  for
a
                           constant payment loan or the periodic savings for a
future balance.
nomrate       (financial)  Returns the nominal annual interest rate.
savings       (financial)  Returns the balance of a periodic savings by using
variable
                          interest rates.
timevalue     (financial)  Returns the equivalent of a reference amount at a
base date
                           by using variable interest rates.
```

Internal Functions

Internal functions are called upon and cannot be replaced by function definitions.

Function Name	Definition	Example
dim	Determines the dimensions of a variable.	`x[10] = 1;` `y = dim(x);`

newtable	Creates a new table.	`columns = {”make”, “MSRP”, “HP”, “cylinders”};`
		`coltypes = {"varchar", "int64", "integer","int64"};`
		`row1 = {"dodge", 20000, 250, 4};`
		`row2 = {"ford", 30000, 200, 6};`
		`table = newtable( "cars", columns, coltypes, row1, row2);`

add_table_attr	Adds attributes to a table.	`rc = add_table_atrr(table, “Actionset”, “imstat”, “Action”, “Summary”);;`

discard	Discards the variables that are listed as arguments.	`x[10] = 1;`
		`y = 2;`
		`discard(x,y);`

tabcolumns	Gets the types for a table.	`coltypes = tabtypes(table);`

Function Name	Definition	Example
printtable	Prints table to a specified location.	`printtable (table, “<fileref>”, columns);`
addrow	Adds row to a table.	`addrow(table, row1);`
findtable	Searches a given value in the table.	`table = findtable(result);`
traceback	Returns a string consisting of traceback from the current function.	`tb=traceback();`
dictionary function	Returns the value of a key from a dictionary. Determines if the key exists in a dictionary.	`y=dictionary(dict, ”name”);` `y=exists(dict,”name”);`

Math Functions

Function Name	Definition	Example
min	Minimum of a list of values	`y = min(a, b, c, d, e);`
max	Maximum of a list of values	`y = max(a, b, c, d, e);`
sqrt	Square root of a value	`y=sqrt(z);`
lgamma	Natural logarithm of a gamma function	`y = lgamma(x);`
exp	Value of e (the base of natural logarithms)	`y = exp(x);`
log	Function that returns the natural logarithm of x	`y = log(x);`
log2	Base 2 logarithm	`y = log2(x);`

Exceptions

You can define exception-handler functions. All of the exception-handler functions take exactly one parameter which is a dictionary of information. Within the dictionary the context of the exception and parameters are specified.

This is an example of a handler for a system exception such as a floating-point error.

```
function myfphandler(env) do;
put "error detected, using_MISSING_instead";
```

```
setexceptvalue(_MISSING_);
resume;
end do;
on_fpexception call fphandler;
```

The example above is telling the handler when you find an error use "_MISSING_". If SAS finds an error or a floating-point error in your data then it will display MISSING every time there is an error; otherwise, it will continue with the value that it is supposed to display.

Results: CAS Procedure

Result Table

The CAS result table is a table that is created as the result of an action. In addition to rows and columns, the table also contains labels and variable types. The table is the primary means to return information to CASL.

CASL enables you a variety of operations on the CAS result table:

- Reference particular row and column values within a result table.
- Extract a row into a dictionary.
- Extract the column names and types into a dictionary.

CASL enables you to create your own result table from the existing result table or as a subset or combination of other result tables.

Examples: CAS Procedure

Example 1: Set-up Program for PROC CAS

The following example shows you how to access your data, load the data into CAS, and perform any additional setup that is needed to run CASL statements. This example assumes that you have operating system level access to the server. You must be able to save the input data files in the **path-to-data-directory**. For more information on acessing a client-side data, see "Load a Client-Side File" in *SAS Cloud Analytic Services: Accessing and Manipulating Data*

Program

```
options cashost="cloud.example.com" casport=5570; /*1*/
cas casauto; /*2*/
proc cas; /*3*/
    table.addCaslib / /*4*/
    caslib="casdata"
    datasource={srctype="path"}
    path="path-to-data-directory";
run;
```

```
table.loadTable/ /*5*/
    path="iris.sashdat"
    casout={name="IRIS", replace=true};
run;
quit;
```

1. The CASHOST= option specifies the host name of the server to connect to. The CASPORT= option specifies the network port number to connect to.
2. Connect to SAS Cloud Analytic Services and start a session.
3. The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.
4. Set the active caslib. The table.addCaslib action adds a caslib that has access to your data. The input data must be located in a directory that is accessible to the server.
5. Load the input data set **iris.sashdat** into CAS and create an in-memory table called Iris. Define parameters using braces { }.

Example 2: Run an Action

This example runs the specified action, in this case, the listNodes action and displays the contents of the table to the Output Delivery System (ODS). For information about how to download and access the data, see Set Up Program for PROC CAS on page 29.

Program

```
proc cas; /*#1*/
    action listnodes results=res submit; /*#2 */
    put res; /*#3 */
run;
```

1. The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.
2. The ACTION statement runs the specified action, in this case, the listNodes action. The results of the listNodes action are stored in the variable named *Res*. The SUBMIT option specifies to run the action synchronously as programming statements are evaluated. The listNodes action does not accept any parameters, so no parameters are specified.
3. The PUT statement displays the contents of the variable *Res*. The listNodes action returns a table and the PUT statement supplies the table to the Output Delivery System (ODS) for display.

The following output shows sample results for eight machines.

Output 1.2 *Output: HTML Output for listNodes action*

Name	Role	Connected	Server Port	Primary IP Address
rdcgrd001	controller	1	58475	10.124.3.9
rdcgrd002	worker	1	0	
rdcgrd003	worker	1	0	
rdcgrd004	worker	1	0	
rdcgrd005	worker	1	0	
rdcgrd006	worker	1	0	
rdcgrd007	worker	1	0	
rdcgrd008	worker	1	0	

Example 3: Define a New Function

This example defines a new function and executes the function using the IF-ELSE/THEN, DO, and PRINT statements. This example also uses an internal function, *factorial*, to define and execute the newly created function *x*.

Program

```
proc cas; /* #1 */
   function factorial(x); /* #2 */
     if (x < 1.0) then return(x); /* #3 */
     else do;
        return exp(lgamma (x+1));
     end; /* #4 */
   end;

   function factorial(x);
       if (x <= 2.0) then return(x);
       else return( factorialr(x-1)*x);
   end;
run;

   do i = 1 to 9; #5
      x = factorial(i);
      print "   Factorial (" i ") = " put(x,best9.); /* #6 */
   end;
   x = factorialr(75);
        print "Factorial of 75 = " x;
run;
   x = factorialr(75.0);
        print "Factorial of 75.0 = " x;
run;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FUNCTION statement creates a new function that takes one argument.

3 This function uses the **lgamma** and **exp** internal functions to calculate the factorial of a number. The IF-THEN/ELSE statement asks the function *x* to find values where the value of *x* is less than 1.0, if it is then return the value of *x*. The ELSE statement asks to return the run-time math expression function lgamma to be multiplied by the value of *x* and added 1.

4 The END statement ends the function processing.

5 The DO statement iterates over the list from 1 to 9.

6 The PRINT statement prints the results. Put acts as a DATA step.

Log 1.1 *Output Log: Define A New Function*

```
Factorial (1 )  =         1
Factorial (2 )  =         2
Factorial (3 )  =         6
Factorial (4 )  =        24
Factorial (5 )  =       120
Factorial (6 )  =       720
Factorial (7 )  =      5040
Factorial (8 )  =     40320
Factorial (9 )  =    362880
Factorial of 75 =         0
```

```
Factorial of 75.0 = 2.480914E109
```

Example 4: Use Run-time Math Functions

Program

In this example we not only create a new function but also use built-in run-time math functions to calculate the probability of the number of people who share the same birthday in the same room. This example uses built-in internal functions *lgamma* and *exp* to calculate the factorial of a number.

```
proc cas; /* #1 */
    function SharedFeature (feature,number) /* #2 */
        if p = exp(lgamma (feature+1)- lgamma(feature-number+1)
            - number*log(feature)); /* #3 */
        else return (1-p); /* #4 */
    end; /* #5 */
        do n over {3 10 22 23 50 75}; /* #6 */
        p = SharedFeature(365,n) /* #7 */
    print "Chance at least 2 out of " put(n,best3.)
        " share same birthday = " put(p,best8.4); /* #8 */
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FUNCTION statement creates a new function. This function contains two arguments and calculates the probability that at least 2 items share the same feature given *number* of items and *features*.

3 The IF-THEN/ELSE statement executes statements that meet specific conditions. This function uses the **`lgamma`** and **`exp`** internal functions to calculate the factorial of a number.

4 ELSE returns the value of one subtracted from the value of *p*.

5 The END statement ends the function processing.

6 The DO statement iterates over a list.

7 The variable *p* is defined here. Use the function **`SharedFeature`** to calculate the probability that at least two people have the same birthday given the number of people in the room.

8 The PRINT statement prints the results. Put acts as a DATA step.

Note: Functions can be recursive and they can execute any CASL statement. Functions can also be re-defined.

Log 1.2 *Output: Log: Probability of Two People Sharing the Same Birthday in the Same Room*

```
Chance at least 2 out of    3 share same birthday = 0.0082
Chance at least 2 out of  10 share same birthday = 0.1169
Chance at least 2 out of  22 share same birthday = 0.4757
Chance at least 2 out of  23 share same birthday = 0.5073
Chance at least 2 out of  50 share same birthday = 0.9704
Chance at least 2 out of  75 share same birthday = 0.9997
```

Example 5: Develop a Scorecard Using Actions

This example shows you how to create a scorecard from data using the DATA step and CASL statements and actions.

Program

```
data golf;/* #1 */
   format outlook $8.;
   format temperature    best10.;
   format humidity    best10.;
   format windy   $5.;
   format golf    $10.;
   input outlook $ 1-8 temperature humidity windy $ 16 - 21 golf $22 - 32;
   cards;
sunny    85 85 false Donot Play
sunny    80 90 true  Donot Play
overcast 83 78 false Play
rain     70 96 false Play
rain     68 80 false Play
rain     65 70 true  Donot Play
overcast 64 65 true  Play
sunny    72 95 false Donot Play
sunny    69 70 false Play
rain     75 80 false Play
sunny    75 70 true  Play
overcast 72 90 true  Play
overcast 81 75 false Play
```

```
rain     71 80 true  Donot Play
;
run;

cas mySession;
libname cas sasioca sessref=mySession;

data cas.golf;
    set golf;
run;
proc cas; /* #2 */
    loadactionset "casdt";/* #3 */
run;
    gbtreetrain result=r //* #4 */
        table={name = "golf"}
            inputs={"outlook", "windy", "humidity",
            "temperature"}
            target="golf"
            maxlevel =4
            maxbranch=2
            nbins    =5
            binorder =1
            ntree=3
            seed=1234
            code={labelid=999, comment=true, tabForm=true}
    ;
run;
    saveresult r['CodeGen'] replace file="C:\gbt_scorecode.sas";/* #5 */
run;
    file log; /* #6 */
    loadactionset "dataStep";
    code = readpath("C:\gbt_scorecode.sas");/* #7 */
    dscode = "data golf_out;set golf;" || code ||"run;";/* #8 */
run;
    runCode code = dscode; /* #9 */
run;
    fetch / table = {name = "golf_out"};/* #10 */
run;
quit;
```

1 Generate, save, and run DATA step score code using CASL. The DATA step in CAS operates on CAS tables. Input and output data sets must use the CAS LIBNAME engine. The engine enables the DATA step to fetch column metadata for CAS tables when compiling the program. Data sets that use other engines must be loaded into CAS, or a caslib must be defined for that data source. For more information, see "The DATA Step in SAS Viya" in *SAS Cloud Analytic Services: Accessing and Manipulating Data*.

2 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

3 The LOADACTIONSET loads a SAS Cloud Analytic Services action set. Some actions are available as platform-level functionality. You can use the LOADACTIONSET statement to load more action sets in your session. In this example you are loading the actionset 'casdt'.

4 Train model and generate score code. The action gbtreeTrain creates a series of decision trees that together form a single predictive model using the gradient

boosting method.For more information on the syntax of gbtreeTrain, see "Train gradient boosting tree" in *SAS Cloud Analytic Services: Analytics Programming Guide* .

5 Save the DATA step score code to a SAS data set. The SAVERESULT statement creates a SAS data set from the results of an ACTION.

6 The FILE statement enables you to specify different locations for output. In this example we specified the log as the location for our output.

7 Read the file and put the content to a CAS variable using the readpath action. The readpath action reads the contents of the file given into the variable as a string.

8 Concatenate the string of data from the readpath action using | |.

9 Call the runCode action to run DATA step score code. The runCode action runs user-defined code on time series.

10 Fetch rows from a table. Use the format parameter to apply formats to the variables. Use sortBy to specify the variables and variable settings for sorting results. Use the parameter to specify the ordinal position of the last row to return.

Output 1.3 *Results: Generated Score Card*

The SAS System

Results from dataStep.runCode

CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS(casuser)	golf	14	5

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS (casuser)	golf_out	14	7

The SAS System

Results from table.fetch

Selected Rows from Table GOLF_OUT							
Index	outlook	temperature	humidity	windy	golf	GBT_golf	GBT_golf_PredP
1	sunny	85	85	false	Donot Play	Play	0.572737337
2	sunny	80	90	true	Donot Play	Play	0.572737337
3	overcast	83	78	false	Play	Play	0.6804587332
4	rain	70	96	false	Play	Play	0.6124379015
5	rain	68	80	false	Play	Play	0.6804587332
6	rain	65	70	true	Donot Play	Play	0.6804587332
7	overcast	64	65	true	Play	Play	0.6804587332
8	sunny	72	95	false	Donot Play	Play	0.572737337
9	sunny	69	70	false	Play	Play	0.6436719811
10	rain	75	80	false	Play	Play	0.6804587332
11	sunny	75	70	true	Play	Play	0.6436719811
12	overcast	72	90	true	Play	Play	0.6124379015
13	overcast	81	75	false	Play	Play	0.6804587332
14	rain	71	80	true	Donot Play	Play	0.6804587332

Example 6: Computing Partial Dependency

This example uses decision tree to train a regression model. The example takes one variable and integrates other variables by creating a fake data set (_partial_data) and uses the datastep action and dtreescore to score the data set and run a summary to get the predicted response.

Program

```
proc cas ; /* #1 */
    file log; /* #2 */
    datastep.runcode/ code=' /* #3 */
data simuData;
do i=1 to 100; /* #4 */
    x1= rand("uniform");
    x2= rand("uniform");
    x3= int(4*rand("uniform"));
    p= 1/(1+exp(-(sin(3*x1)-4*x2+x3)));
    y=x1*x2+x3;
    output;
end; /* #5 */
drop i;
run;
';
run;

quit;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FILE statement enables you to specify a different location on output. In this example, the FILE statement specifies the log as the location on output.

3 The datastep.runCode action runs a DATA step program to this action as a string.

4 The DO statement iterates over the list. In this example, the DO statement identifies the loop variable as *i* and loops from 1 to 100.

5 The END statement ends the DO processing.

Log 1.3 *Output: Log*

```
NOTE: Active Session now TESTSESS.
 CAS Library          Name  Number of Rows  Number of Columns
CASUSERHDFS(sabisw)  simuData    444800          5
```

```
/* Train a Simple Regression Tree*/
proc cas ;
    decisionTree.dtreetrain result=r /table={name='simuData'} /* #1 */
            inputs=${x1-x3}
            target='y' casout={name='mytree',
        replace=1}
            nbins=50 maxbranch=2 maxlevel=5 binorder=true
        mergebin=true leafsize=1;
    print r; /* #2 */
run;

quit;
```

1 The action set decisionTree calls on the action dtreetrain to train a decision tree for classifications or regressions.

2 The PRINT statement prints the results.

```
proc cas ;
    /* Let's compute the partial dependency for one point.*/
    function partialDependencyForOnePoint(inputTableName, modelName, /* #1 */
        partialVarName, colName, debug);
    compExpr=partialVarName || '=' || colName || ';'; /* #2 */
    if (debug==1) then print compExpr; /* #3 */
    decisionTree.dtreescore result=r/table={name=inputTableName,
        compvars=partialVarName, comppgm=compExpr}  /* recreate original x */
        model={name=modelName}

                                        casout={name='_scored_out', replace=1}
                                path=0;
    if (debug==1) then print r;

    /* get the predicted y */
    simple.summary result=r/ table={name='_scored_out', vars='_DT_PredMean_'}; /* #4 */

    /* drop this table */
    if (debug==0) then do;
    droptable/table='_scored_out';
    end;
    return r['summary'][1, 'mean'];
    end func;

    /* inputTableName: input table */
```

```
    /* modelName: model name for dtreescore */
    /* partialVarName: the variable name to be computed for dependency */
    /* inputVarNameList: the variables need be integrated out */
    /* lowerBound: lower bound of partialVarName */
    /* upperrBound: upper bound of partialVarName */
    /* nPoints: number of points */
    /* debug: if 1, enable debug mode */
    function partialDependency(inputTableName, modelName, partialVarName,
        inputVarNameList, lowerBound, upperBound, nPoints, debug);

    /* generate a new set of column names */
    newColumns={};
    do i=1 to nPoints;
    newCol=partialVarName || '_' ||(string)i;
    newColumns=newColumns + $(newCol);
    end;
    codeExpr="data _partial_data;"
                ||"set "||inputTableName||";"
                ||"keep ";
    nInputVars=dim(inputVarNameList);
    do i=1 to nInputVars;
    codeExpr=codeExpr || inputVarNameList[i] || ' ';
    end;
    do i=1 to nPoints;
    codeExpr=codeExpr || newColumns[i] || ' ';
    end;
    codeExpr=codeExpr || ';';
    myx=lowerBound;
    width=(upperBound-lowerBound)/nPoints;
    do i=1 to nPoints;
    codeExpr=codeExpr || newColumns[i] || '=' || (string) (myx+(i-1)*width) ||';';
    end;
    codeExpr=codeExpr || 'output;';
    if (debug==1) then print codeExpr;
    datastep.runcode result=r / code=codeexpr;

    /* now we get a wide table and then run each scoring action
for each selected column */
    myx=lowerBound;
    do i=1 to nPoints;
    x=myx+(i-1)*width;
    y=partialDependencyForOnePoint('_partial_data', modelName, partialVarName,
        newColumns[i], debug);
    res.x[i]=x;
    res.y[i]=y;
    end;

    /* drop the temp table */
    if (debug==0) then do;
    droptable/table='_partial_data';
    end;
    return res;
    end func;
run;

/* let us try it by integrating x2 and x3 out */
```

```
otherVars={'x2', 'x3'};
res=partialDependency('simuData', 'mytree', 'x1', otherVars, 0, 1, 10, 0);

/* print result into the log */
print res;
run;
```

1 The FUNCTION statement creates a new function named partialDependencyForOnePoint and has five arguments named inputTableName, modelName, partialVarName, colName, and debug.

2 A new variable is created named *compExpr*. This variable is created using the Assighment statement. For more information see"ASSIGNMENT Statement" on page 7.

3 The IF statement executes statements that meet specific conditions. In this example, if the argument debug equals to 1 then using the PRINT the value of the variable *compExpr*.

4 The action set simple executes the action summary to generate descriptive statistics about the data and the results are saved into a new result variable named *r*.

Log 1.4 *Output: Log: Output from Variable Res*

```
{x={0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9},y={1.6166262987,1.6166262987,1.6166262
987,1.6480508194,1.6480508194,1.6635301729,
 1.8676608205,1.8676608205,1.8676608205,1.8676608205}}
```

Output 1.4 *Results: Computing Partial Dependency*

r: Results from decisionTree.dtreeTrain

Decision Tree for SIMUDATA	
Number of Tree Nodes	31.000000
Max Number of Branches	2.000000
Number of Levels	5.000000
Number of Leaves	16.000000
Number of Bins	50.000000
Minimum Size of Leaves	26522
Maximum Size of Leaves	30175
Number of Variables	3.000000
Alpha for Cost-Complexity Pruning	0
Number of Observations Used	444800
Maximum STD of Leaves	0.155561
Minimum STD of Leaves	0.052769
Mean Squared Error	0.013448

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS(casuser)	mytree	31	23

Recommended Reading

- *Getting Started with CASL*
- *SAS Cloud Analytic Services: Analytics Programming Guide*
- *SAS Cloud Analytic Services: Accessing and Manipulating Data*
- *SAS Cloud Analytic Services: Data Mining and Machine Learning Programming Guide*
- *SAS Cloud Analytic Services: Statistics Programming Guide*
- *SAS Cloud Analytic Services: System Programming Guide*

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books