

# **NASIS CVIR Language Manual**

Scripting language for NASIS **C**alculations, **V**alidations, **I**nterpretations and **R**eports

NASIS 6.1 Edition

US Department of Agriculture  
Natural Resources Conservation Service

Gary Spivak  
February 1, 2011

# Contents

NASIS CVIR Language Manual .....	1
Overview of CVIR Scripts.....	1
Data Flow in CVIR Scripts.....	2
Query Scripts .....	3
Property Scripts .....	3
Calculation and Validation Scripts.....	4
Report Scripts .....	5
Text Style Reports.....	6
XML Style Reports .....	6
Running Reports Against Local or National Database.....	7
CVIR Syntax Reference .....	9
Conventions used in this Guide .....	9
ACCEPT.....	10
BASE TABLE .....	11
DEFINE .....	12
Storing Multiple Values in a Variable .....	13
Expression Syntax.....	13
Explanation of Expression Syntax .....	17
String Expressions .....	18
Function Expressions .....	21
Numeric Functions .....	27
REGROUP Expression.....	31
DERIVE.....	34
EXEC SQL .....	35
EXEC SQL: Sort Specification.....	39
EXEC SQL: Aggregation Specification .....	40
FONT .....	46
HEADER and FOOTER.....	47
INPUT .....	48
INTERPRET .....	49
Using Interpretations in Reports: .....	50
MARGIN.....	52
PAGE .....	53
PARAMETER .....	54
PITCH .....	58
SECTION .....	59
SECTION: Conditions .....	61
SECTION: KEEP option .....	63
SECTION: Output Specifications.....	64
AT Statement .....	66
ELEMENT Statement .....	68
Column Specifications.....	72
Column Layout Specifications .....	74
SET .....	79
TEMPLATE .....	80
WHEN.....	81
Using Subreports .....	82
Appendix 1: Conventions for Web Soil Survey Reports .....	82

DocBook XML .....	82
XML Elements Used in Reports .....	83
Elements used in tables .....	86
Elements used in non-table reports .....	86
Attributes used in all elements .....	87
Parameters for Reports.....	89
Using parameters in a report query .....	90
Script Variables.....	91

---

## Overview of CVIR Scripts

In NASIS, all Queries, Reports, Calculations, Validations and Properties contain a *script*, which is a set of instructions for reading data from the database and using the data to produce some result. All these scripts have a common structure with various options that are specific to their function. This reference manual contains the complete specifications for CVIR scripts.

The major parts of a CVIR script are:

- **Query:** Instructions to read data from the database, written in a variant of SQL (Structured Query Language, an international standard for working with relational databases, often pronounced “sequel”). A simple CVIR query looks like:  

```
EXEC SQL SELECT musym, muname from mapunit, lmapunit
WHERE JOIN mapunit TO lmapunit;
```
- **Data Manipulation:** A series of instructions for working with the data to produce new data values, using mathematical formulas, if-then-else logic, and other operations. An example of a data manipulation statement is:  

```
DEFINE complabel IF ISNULL(localphase) THEN compname
                ELSE compname || ", " || localphase.
```
- **Output:** Instructions for producing some result. In a report script this includes the specifications for laying out the report; for a calculation script it specifies which columns of the database record will be updated; and so forth. Typical output statements for a report are:

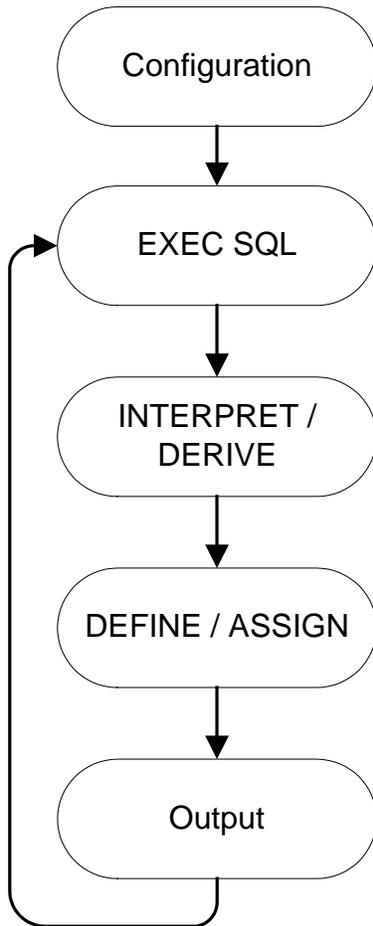
```
TEMPLATE mapunit
  AT LEFT FIELD WIDTH 20, FIELD WIDTH 8 SEPARATOR "|".
SECTION
  DATA
    USING mapunit muname, musym.
END SECTION.
```

CVIR scripts are stored in tables in the NASIS database and are assigned ownership in the same way as data in the soils tables. Anyone can run a CVIR script, but you must be a member of the group that owns the script, and you must check it out from the server, in order to edit it. The scripts are found in the Explorer area (left side) of the NASIS screen, organized by the type of script and the NASIS site. Menu options are provided to run a script or open it for viewing and editing.

The CVIR Syntax Reference section of this document describes each CVIR statement in detail. It is arranged in alphabetic order, with a note on what types of scripts each statement can be used in. That information will help you get the syntax right, but doesn't tell you the overall concepts. The next few sections will give some guidelines on writing different kinds of scripts.

---

## Data Flow in CVIR Scripts



A script specifies a set of actions to be performed but not necessarily the order in which they will be performed. The CVIR processor collects all the statements of a particular kind and processes them as a group at the point in the data flow where they are needed. The sequence is:

1. Configuration statements are processed just once before anything else happens. These are statements that don't produce any actual data, but instead specify conditions for other statements to work with. They include:
  - ACCEPT
  - PARAMETER
  - BASE TABLE
  - PAGE, MARGIN, FONT and PITCH
  - TEMPLATE
2. Data input statements provide the data the script works on. Most scripts include the EXEC SQL statement to get data from the database. The INPUT statement can be used to read data from a file. There are a number of rules about the interactions between data input statements when the script contains more than one, which are described in the Syntax Reference section.
3. References to other scripts are processed next. The DERIVE statement is used to obtain data from a Property script, and the INTERPRET statement generates interpretations that can be displayed in a report.
4. Data manipulation is the next step. This is done with the DEFINE and ASSIGN statements, which can include mathematical formulas, if-then-else conditions, and other functions. The result is a set of variables whose values can be used in the next step.
5. The final step is output of data, and this varies depending on the type of script. In a Report script, the output specifications define how the information will be formatted. For a Calculation script, the output specifications identify what fields in the database will be updated, while a Validation script specifies the messages to be produced when error conditions are found. A Property script has no output section at all.
6. After completion of these steps the process continues with step 2 and repeats until there is no more data to process.

---

## Query Scripts

In NASIS a Query (with a capital Q) is a script used to find data for use in a NASIS session. A Query can be run against the national database to identify data to be downloaded to the local (workstation) database, or it can be run against the local database to identify the data that will appear in the table editors and reports (known as the *selected set*).

A Query script is the most minimal of all CVIR scripts. It consists of exactly one, partial SQL query. It does not even use the EXEC SQL prefix as in other CVIR scripts. The only parts of a SQL query it uses are the FROM and WHERE clauses. When a Query is run, NASIS produces the full SQL query using the script plus any run-time options selected by the user, such as target tables and parameters.

Details on the CVIR variant of SQL are in the Syntax Reference section under EXEC SQL. The key features of a Query script are:

- All the tables listed in the FROM clause are candidates to be *target tables* when the Query is run. For each target table the user picks, a SQL query is constructed to select rows in that table. Additional rows linked to the target table are then found to fill out the selected set or download list.
- Query parameters can be specified in the WHERE clause by using a comparison with a question mark, such as “areasympol = ?”. When the query is run, NASIS creates a field for the user to enter an area symbol. It automatically looks up “areasympol” in the metadata to determine its data type and whether it has a fixed choice list (domain).

## Property Scripts

After the Query, the next simplest type of script is the Property. It has several uses in NASIS. As the name suggests, a Property is primarily a way to get some soil property data from the database. It typically contains all the logic needed to select data from the appropriate tables, aggregate to the required level and perform other needed transformations. Putting all this in a Property script provides consistency in the use of a particular soil property.

Property scripts are used:

- In the Evaluation portion of an Interpretation. In the Evaluation Editor the user picks exactly one Property to be the source of data for the evaluation function. When used for this purpose the Property must define one or more of the variables “low”, “rv” and “high” for use by the Evaluation. The type of data and number of these variables must agree with the Data Type and Modality fields entered on the General page of the Property editor. The Property cannot use the ACCEPT statement to take parameters.
- In the DERIVE statement of another CVIR script (Report, Calculation, Validation or another Property). When used this way you are not limited to the variables “low”, “rv” and “high” because the DERIVE statement specifies which Property variables it is

---

expecting. The **DERIVE** statement can also pass parameters to the Property's **ACCEPT** statement.

- In the Interpretations Editor with the Run button. This is just used for testing a Property. The user first highlights a row in the Property's base table and then clicks Run. A page of output is displayed listing all the variables defined in the Property script and the values they take on when the Property is run with the selected row of data.

Key features of a Property script include:

- There must be a **BASE TABLE** statement to specify for which database table the queries in the script are run. Properties used in Interpretations will have:  
`BASE TABLE` component.
- The script can include one or more queries. The queries are run in the context from which they are called. For example, when called from an Evaluation the queries return data for the Component record being evaluated, and when called from a CVIR script they return data for the row of the Base Table that the caller is operating on. The CVIR engine does this by adding an extra condition to the query's Where clause to limit its results to the current row of the base table.
- The script can invoke other Property scripts using the **DERIVE** statement, as described in the Syntax Reference section.
- The script can include **DEFINE** and **ASSIGN** statements to create new variables or modify the values of variables supplied by the queries.

## Calculation and Validation Scripts

Calculations and Validations are scripts that automate complex data entry or data checking procedures. A Calculation is an approved, standard procedure for deriving data values using data previously entered in the database. A Validation is an approved, standard procedure for checking that various data in the database are consistent with each other and/or fully populated.

To use a Calculation or Validation script:

- Open the Table Editor to the Base Table defined in the script.
- Select one or more rows in the table.
- Locate the Calculation or Validation in either the Explorer or Editor window and click on Run.
- If running a Calculation, check the options as needed to allow the calculated results to replace data that was manually entered (tagged M) or entered prior to the time that source tracking was established (tagged P). If these are not checked a Calculation will only replace empty fields or ones with previously calculated values.

---

Key features of Calculation and Validations scripts include:

- The script can contain all the parts described above for Property scripts.
- A Validation script has one or more [WHEN](#) statements which produce messages when a specified condition occurs. The messages are displayed in the NASIS message window, along with a link to the row that produced the message.
- A Calculation script has one or more [SET](#) statements that identify the fields where calculated results will be placed. A Calculation script can also use **WHEN** statements to produce message about problems with the calculation.
- Calculations and Validations may be created or edited only by people belonging to an administrative (NSSC Pangaea) group in NASIS. If the script is marked Not Ready for Use, only a member of the owning group can run the script.

## Report Scripts

A Report is the most complex type of CVIR script because there are many details that have to be specified to produce the exact output format that you want to see. There are also two independent styles of report generation, text based and XML based, which will have to be discussed separately. And finally, there is an option to run a report on either the local or the national database, which can produce different results.

To use Reports:

- Open a Report from the Report Explorer to see the script and other details in a Report Editor window.
- Run an open Report from the Report Editor, or run a Report without opening it by selecting it in the Report Explorer and picking a Run option from the Explorer menu. Select either Run Against Local Database or Run Against National Database. See notes below about running against the national database to avoid producing too much output.
- The output of a Report is placed in a temporary file on your computer and then opened with the standard application for the type of file. For example, the standard application for HTML output is a web browser (such as Internet Explorer), and for text (TXT) output the application might be Notepad. You can configure the application for each file type (in Windows XP) using Control Panel / Folder Options / File Types.

Key features of a Report script include:

- The script can contain all the parts described above for Property scripts, plus output specifications.

- 
- A script that contains nothing but a query (EXEC SQL statement) produces a standard report output consisting of a simple table with each field of the SELECT list as a column. This is convenient for a quick-and-dirty type of data listing.
  - The INTERPRET statement can be used to generate interpretations for inclusion in the report output.
  - For text style reports the default layout is an 8.5 by 11 inch page with ½ inch margins and 12 point font. Page layout statements can be used to change these specifications. For XML or HTML style reports no page specifications are used, because the browser and style sheet control the final appearance of the report.
  - The TEMPLATE statement can be used to define the layout of a line of report output. This makes it easy to apply the same layout to many output lines, to produce a consistent appearance.
  - The SECTION statement provides a way to group the report output statements and specify when they will be used. A Section represents a block of report output that can appear at the beginning or end of the report, at a point where some data changes, or repeatedly.

## Text Style Reports

Text style output is the type of report produced by NASIS 5.x and earlier. It consists of lines of text, where a line is just a series of characters and each character takes up a fixed amount of space. When a series of lines with consistent layouts are produced, the characters line up to produce the appearance of columns in a table. Often a vertical bar symbol ( | ) is used to separate report columns. Text style output generally does not look right when printed with a proportional font.

Text output does not have to be in table format. It can look like paragraphs or lists, since each line of output can have its own format. The output can also be stored in a file and imported into another program, such as a spreadsheet, using either a fixed column width or variable width and delimiters. There are existing reports in NASIS that can be used as examples for any of these output formats. It is much easier to copy and modify an existing report than to start from scratch.

The [AT](#) command is the key to producing text style output. It specifies where on the line each piece of data will be placed, and how it will be formatted. There are numerous options, described in the reference section below.

## XML Style Reports

XML style output was developed for Web Soil Survey and has been included in NASIS 6. XML (eXtended Markup Language) is an example of a standard data format called “markup” because it contains markings, or tags, mixed with the text. A tag is a name, and possibly some other attributes, enclosed in angle brackets, such as <table>. A tag typically describes how the

---

following text is used, or what type of text it is. XML output can be used in a variety of ways, including an import to other programs or a source for transforming into formatted reports.

XML output is produced with the [ELEMENT](#) command. A report cannot use both `ELEMENT` and `AT` commands. An `ELEMENT` always has a name, and may also have attributes and content:

- An element name must correspond with a standard XML tag for the type of output you want to produce. For example, a formatted table begins with a `<table>` tag, which is specified in a NASIS report as `ELEMENT "table"`. The standard tags are listed in Appendix 1.
- Many XML tags have standard attributes that modify the output appearance. For example, a table can have borders drawn between cells with a tag like `<table border="1">`. In a NASIS report this is written as  
`ELEMENT "table" ATTRIB("border", "1").`
- Almost all XML tags have some content, which could be other tags or just data to be output. The content is preceded with a tag like those just described, and followed with a closing tag that has just the tag name prefixed with a slash. For example, a paragraph in XML might be `<para>This is some information to print.</para>`. The corresponding NASIS statement would be  
`ELEMENT "para" "This is some information to print."`
- A plain `ELEMENT` command produces both the opening and closing tags. Sometimes it is not possible to put all the content you want into a single `ELEMENT` command, so you can use `ELEMENT OPEN "name"` to produce just the opening tag. Later in the report script you must use `ELEMENT CLOSE "name"` to produce the closing tag.

NASIS allows the report output format to be specified as HTML, which is just a variation of XML. To produce reports that look like Web Soil Survey reports, use the elements and attributes listed in the appendix for the DocBook XML standard. This is converted automatically to HTML for displaying in a web browser. If you are familiar with HTML you can use regular HTML tags as NASIS elements instead of DocBook XML.

## Running Reports Against Local or National Database

When a report is run against the local database it normally uses only data in the *selected set*. This is convenient because a report can be designed without parameters and simply run with different selected sets to get different output. But it also means that Queries need to be run to get data into the selected set for every table used in the report.

A report can also be written to use any data in the local database, regardless of whether it is in the selected set. To do this, put the word `REAL` before the table name in the `FROM` clause of the report query. A query can have a mix of `REAL` tables and standard tables.

When a report is run on the national database there is no selected set, so all tables act as if they had `REAL` on them. Since the national database is very large, it is important to plan reports

---

carefully to avoid excessive run time and output. A report designed to work on the selected set is probably not appropriate for the national database. The **PARAMETER** statement should be used so that the user can specify criteria to limit the report output.

Reports can be run on the national database either normally or offline. When a report is run normally the program waits for the report to complete then displays the output. With an offline report, you send the request in to the national database and continue working in NASIS. When the report completes you will receive an email with a link to the report output. Running national reports in the normal way is subject to time limitations and the reports sometimes don't finish. Off line reports are allowed to run longer.

---

## CVIR Syntax Reference

### Conventions used in this Guide

CVIR script statements are arranged alphabetically in this technical guide so you can find them more easily. The description consists of the following parts:

**Syntax:** The syntax is described in a formal notation, using the following conventions:

- Braces { } enclose a set of alternatives, of which one must be chosen.
- Any portion of a definition in brackets [ ] is optional.
- When an ellipsis (...) follows the brackets, the optional part can be repeated.
- The symbol  $\Rightarrow$  means “is defined as”, and defines a term that appears in a previous statement definition.
- Punctuation in bold print is a required part of the statement.
- Keywords in the CVIR language are shown in sans-serif capital letters (like **DEFINE**), but the interpreter is not case sensitive for keywords or variable names.

**Used In:** The Used In line identifies the type of scripts in which the statement may be used. The types are Report, Subreport, Property, and Calculation (Validation scripts use the same statements as Calculations).

**Example:** One or more examples of the statement are shown, followed by an explanation.

**ACCEPT****Syntax:**

```
ACCEPT variable [ , variable ] ... .
```

```
variable ⇒ name
```

**Used In:**

**Subreport, Property, Calculation**

**Example:**

```
ACCEPT datamapunit_iid.  
ACCEPT top_limit, bottom_limit.
```

The **ACCEPT** statement defines variables that are passed into the script. These variables can be used in expressions to calculate values for other variables. They can also be used in the **WHERE** clause of a query by writing **\$name**, where **name** is the name of the variable. This creates a parameterized query, as discussed under [EXEC SQL](#).

The first example of the **ACCEPT** statement could be used in a subreport. The value of a key column such as **datamapunit\_iid** might be passed by a higher level report, and the subreport would use it in a query to find data related to the data mapunit being processed in the higher level report.

The second example might be used in a Property script. In this case two variables are passed by some script that calls the property, and could be used to perform some calculation in the property script. Any variable names may be used because they do not refer to database columns.

The number of parameters passed by the caller must equal the number of variables in the **ACCEPT** list. The type and dimension of these variables are not predefined, so they are determined by the values passed by the caller. For a Property, the primary key column of the base table acts as if it were in the **ACCEPT** list, even if the property script has no **ACCEPT** statement. It is used to ensure that the property is providing data for the same record as the script that calls it. Consequently, a calling script and its called property scripts must use the same base table.

**BASE TABLE****Syntax:**

BASE TABLE *table-name* .

**Used In:****Report, Property, Calculation****Example:**

BASE TABLE component .

When a CVIR script and its associated properties have several database queries, automatic coordination is performed between the queries by specifying a **BASE TABLE**. For example, if the base table is Component, the automatic coordination assures that each query provides data for the same component during a cycle of the script. A report script requires a base table if the script includes an **ACCEPT** statement, more than one query, or **DERIVE** statements.

In a report script the processing cycle is determined by the aggregation specifications in the first report query, but the base table provides the key used to synchronize queries and properties. For example, the first query may include a statement like:

**AGGREGATE ROWS** muiid, coiid

Here the component id (coiid) is the lowest level of aggregation, so the report performs a cycle for each component. Normally the base table for this report, if needed, would also be component. Deviation from this norm is an advanced report capability which requires careful testing to see that the report works correctly.

A Calculation/Validation script differs from a report in that it performs one complete execution for each base table row that has been selected by the user and has been checked out for editing. It accepts key input values from the current base table row, and stores its calculated data elements in the same row.

A Property script requires a **BASE TABLE** to coordinate its query with those in the calling script. It does not need a **BASE TABLE** if it uses a parameterized query

The *table-name* used in the **BASE TABLE** statement must be one of the tables defined in the NASIS metadata. Either the logical name or the physical name may be used, but the name must be spelled the same as in the queries.

## DEFINE

### Syntax:

DEFINE variable [ expression ] [, expression ]... [ initialization ] .

ASSIGN variable expression [, expression ]... .

expression (*see next page*)

initialization (*see next page*)

### Used In:

#### Report, Property, Calculation

### Example:

```
DEFINE status CODENAME (mustatus).
ASSIGN status status || " mapunit".
```

The **DEFINE** statement defines a variable for use in a CVIR script. Each defined variable name must be unique within a script, and must be different from the names of columns in the input queries. The **ASSIGN** statement recalculates the value of a variable that was defined in a previous **DEFINE**, **DERIVE**, or **EXEC SQL** statement.

Names of variables may be any combination of letters, numbers and the underscore character provided that the name starts with a letter and is not the same as one of the reserved words in the language. Reserved words are printed in sans-serif **CAPITALS** in this document. Different scripts may use the same variable names but the variables are independent, even if one script calls another via a **DERIVE** statement. There is one restriction on variable names for Properties that are used in interpretations. Evaluations take their input data from variables named “low”, “rv”, and “high” (or just “rv” if the property modality is RV). A Property called by an Evaluation can use other variables for intermediate results, but has to place its final results in variables with these names.

Each variable is given a value by an expression or a list of expressions separated by commas. An expression may be based on literals, columns from the input, or other variables. When a list is used, the listed values are combined into an array that has as many values as the sum of the number of values of the items in the list. On each cycle of the input, all variables are recalculated in the order that they appear in the **DEFINE** and **ASSIGN** statements. Variables are not explicitly typed, so the data type is determined by the result of the expression.

An initial value for a variable can also be specified in the **DEFINE** statement. A variable defined with an initial value and no expression is simply a constant; its value will not be changed. Only a single initial value can be specified, not a list of values.

An important use for an initial value is with an expression that contains the variable being defined. Consider the statement: **DEFINE** list (list || name) **INITIAL** "Names: ". This takes the column "name" from each input record and concatenates it to the variable "list", following the initial string "Names: ". If no initial value is defined with this type of expression, the variable starts out with a null value, which could produce undesirable results.

### **Storing Multiple Values in a Variable**

A variable may hold a single value or multiple values. The number of values that a variable holds is called its *dimension*. Multiple valued variables are sometimes referred to as *arrays*. They can be created in a number of ways:

- With an **AGGREGATE** clause in an **EXEC SQL** statement.
- From a Property called by a **DERIVE** statement.
- From a list of values or expressions in a **DEFINE** statement.
- By using the **APPEND** operator.

Depending on the aggregations and other operations used, the variables in a report can end up with different dimensions. Some operators, such as **LOOKUP** and **WTAVG**, can cause a report to fail if the dimensions of their arguments are not the same. So attention must be paid to the way multiple valued variables are processed.

Most of the operators used in expressions do not change the dimension of the data. If an operator uses two or more variables of different dimensions, the result will generally have the largest dimension of the arguments. For example, multiplying a variable with values (1,2,3,4) by the single value 5 produces the multiple valued result (5,10,15,20).

A query that finds no rows results in variables with a dimension of 0, which are typically treated the same way as null values. If all the arguments to an operator have dimension 0 the result will also have dimension 0, but if there is a mixture of zero and non-zero dimensions, the result has the larger dimension. In the above example, multiplying the array (1,2,3,4) by a variable with no values would produce an array of four nulls.

Operators that do not follow these rules are noted in the individual descriptions below. Examples are the array operators like **ARRAYSUM** that reduce an array of values to a single value.

### **Expression Syntax**

The following syntax rules define all the types of expressions that may be created.

expression  $\Rightarrow$  {  
 literal  
 element  
 variable  
 arithmetic\_expression  
 conditional\_expression  
 boolean\_expression  
 string\_expression  
 regroup\_expression  
 function  
 ( expression )

initialization  $\Rightarrow$  INITIAL literal

literal  $\Rightarrow$  { *number* | "*string*" }

arithmetic\_expression  $\Rightarrow$  { expression { + | - | \* | / | \*\* } expression  
 - expression }

conditional\_expression  $\Rightarrow$  { expression ? expression : expression  
 [ IF ] expression THEN expression ELSE expression }

boolean\_expression  $\Rightarrow$  {  
 comparison  
 ISNULL ( expression )  
 NOT boolean\_expression  
 ANY expression  
 ALL expression  
 ( boolean\_expression )  
 boolean\_expression { AND | OR } boolean\_expression

comparison  $\Rightarrow$  expression {  
 ==  
 !=  
 <  
 >  
 <=  
 >=  
 MATCHES  
 IMATCHES } expression

string\_expression ⇒ {  
expression [ *number* : *number* ]  
expression || expression  
CLIP (expression)  
UPCASE (expression)  
LOCASE (expression)  
NMCASE (expression)  
SECASE (expression)  
TEXTURENAME (expression)  
GEOMORDESC (expression, expression, expression)  
STRUCTPARTS (expression, expression, expression)  
ARRAYCAT (expression, delimiter)  
REPLACE (expression, expression, expression)  
DATEFORMAT (expression, format)  
NAMECAP (expression, expression, type, expression)  
}

regroup\_expression ⇒ REGROUP expression BY expression  
AGGREGATE aggregate\_function

function ⇒

```

NEW (expression)
CODENAME (expression [, name ])
CODELABEL (expression [, name])
APPEND(expression, expression)
ARRAYSUM (expression)
ARRAYCOUNT (expression)
ARRAYAVG (expression)
ARRAYMIN (expression)
ARRAYMAX (expression)
ARRAYMEDIAN (expression)
ARRAYMODE (expression)
ARRAYPOSITION (expression)
ARRAYSTDEV(expression)
ARRAYSHIFT (expression, expression)
ARRAYROT (expression, expression)
LOOKUP ([ expression, ] expression, expression)
WTAVG (expression, expression)
SPRINTF (" string", expression [, expression ] ...)
TODAY
USER
SUM (expression)
COUNT (expression)
AVERAGE (expression)
MIN (expression)
MAX (expression)
LOGN (expression)
LOG10 (expression)
EXP (expression)
COS (expression)
SIN (expression)
TAN (expression)
ACOS (expression)
ASIN (expression)
ATAN (expression)
ATAN2 (expression, expression)
SQRT (expression)
ABS (expression)
POW (expression, expression)
MOD (expression, expression)
ROUND (expression [, expression])
    
```

### **Explanation of Expression Syntax**

An expression can produce either a numeric or a character string value, depending on its contents. Numeric and character data can be mixed in expressions, and the data will be converted to the appropriate type if possible. If a conversion is not possible (such as trying to convert “abc” to a number) an error message will be produced.

Expressions are evaluated in order of operator precedence, where higher precedence operations are performed before lower precedence operations. For example, the arithmetic expression:  $A + B * C$  is evaluated as  $A + (B * C)$  because multiplication has higher precedence than addition. When two operators of equal precedence are next to each other, the one on the left is performed first. To make the order of evaluation explicit, put parentheses around the part that should be performed first, such as  $(A + B) * C$ .

The operator precedence from highest to lowest is:

- functions
- multiplication and exponentiation (\*, / or \*\*)
- addition and concatenation (+, - or ||)
- comparisons
- Boolean expressions
- conditional expressions

Most of the expressions involving arithmetic, boolean and comparison operators require little explanation. They work as would be expected, and produce numeric results. The operator \*\* denotes exponentiation; the expression  $A ** B$  is equivalent to the function  $POW(A, B)$ . Comparisons and boolean expressions produce a 1 for True and a 0 for false. The MATCHES comparison works as in Informix: a variable can be compared with a pattern string containing wild card characters, with \* matching any string of characters, ? matching any single character, and square brackets [] enclosing a list of characters to be matched. IMATCHES is the same but performs a case insensitive match.

If a null value is used in an expression, the result is normally null. However, in comparisons, a null value is treated as less than any non-null value and two nulls are considered equal to each other. In boolean expressions, a null is considered False. Invalid computations, such as division by zero, produce a null result. Special cases with null values are noted individually.

**String Expressions**

String expressions allow for substring extraction, string concatenation, and case changes. They expect to operate on character type input, and will convert the input to character if necessary. Note that when a number is converted to a string it is expressed with 6 decimal places. To produce different formats for numbers, use the **SPRINTF** function. The results of the following string expressions are always character strings:

***expression [ n1:n2 ]***

Returns a substring of the string expression, starting at position *n1* for a length of *n2* characters. The first character of the string is position 0. Note, this differs from the way substrings are defined in SQL queries.

**Example:** if variable A has the value “Sample”, the expression A[1:3] returns the value “amp”.

***expression || expression***

Concatenates two strings.

**Example:** the expression “ABC” || “DEF” produces the string “ABCDEF”. If one expression in a concatenation is null it is treated as the string “”, so the result is not a null value unless both of the expressions are null.

***CLIP (expression)***

Removes trailing blanks from a string. This is not normally necessary because NASIS removes trailing blanks when reading data from the database.

**Example:** the expression CLIP(“ABC ”) produces the string “ABC”.

***UPCASE (expression)***

Converts a string to upper case.

**Example:** the expression UPCASE(“ABc12”) produces the string “ABC12”.

***LOCASE (expression)***

Converts a string to lower case.

**Example:** the expression LOCASE(“ABc12”) produces the string “abc12”.

***NMCASE (expression)***

Converts a string to “name” case: first letter of each word upper case and the remainder lower case.

**Example:** the expression NMCASE(“now is the time”) produces the string “Now Is The Time”.

### **SECASE (*expression*)**

Converts a string to “sentence” case: first letter of the string upper case and the remainder lower case.

**Example:** the expression SECASE(“now is the time”) produces the string “Now is the time”.

### **TEXTURENAME (*expression*)**

Converts a set of texture codes to a special string format used in reports. The expression used by TEXTURENAME can have zero or more values, each of which is a string used as a code value for the NASIS data element “texture”. This element can contain a mixture of codes for texture classes, modifiers, and terms used in lieu of texture. The codes are expanded and concatenated together, with commas as necessary, to produce a texture description as used in manuscript reports.

**Example:** if the variable T has two values, one of which is “SL”, and the other is “SR- CL GR-SIL”, the expression TEXTURENAME(T) produces a result with two values, the string “sandy loam”, and the string “stratified clay loam to gravelly silt loam”.

### **GEOMORDESC (*expression, expression, expression*)**

Converts data from the component geomorphic description to a standard landform description string for use in reports. The three expressions used as input can be arrays, but all must have the same number of values. The first parameter is the feature name or names for a component, the second has the feature Id for each feature, and the third has the Exists-On reference for each feature. Where an Exists-On reference matches a feature ID, the two names are combined with the word “on”. If two features have the same feature ID the Exists-On reference is attached to both and they are output as separate strings. Other features that do not have an Exists-On relationship are output as separate strings. The number of values in the result can be more or less than the number of values in the input expressions.

**Example:** Data for this operation would be obtained by joining the component geomorphic description table and the geomorphic feature table, such as:

```
EXEC SQL
SELECT geomorph_feat_name, geomorphic_feat_id,
exists_on_feature
FROM component, component_geomorph_desc, real
geomorph_feature
```

```
WHERE JOIN component TO component_geomorph_desc
AND JOIN component_geomorph_desc TO geomorph_feature;
AGGREGATE COLUMN geomorph_feat_name NONE, geomorphic_feat_id
NONE, exists_on_feature NONE.
```

Assume this query produces the data shown in the following table:

geomorph_feat_name	geomorphic_feat_id	exists_on_feat
alluvial fan		
till plain	1	
pothole	2	1

The expression GEOMORDESC(geomorph\_feat\_name, geomorphic\_feat\_id, exists\_on\_feat) would produce a result with two values, “alluvial fan” and “pothole on till plain”.

**STRUCTPARTS** (*expression, expression, expression*)

Converts data from the Pedon Horizon Soil Structure table to a standard structure description string for use in reports. The parameters are used in the same manner as the GEOMORDESC function above. The first parameter would be the type of structure, usually a string concatenated from *structure\_grade*, *structure\_size*, and *structure\_type*. The second parameter is the row identifier, *structure\_id*, and the third parameter is the reference column, *structure\_parts\_to*. The only difference between GEOMORDESC and STRUCTPARTS is that the latter uses the words “parting to” to separate linked structures, instead of “on”.

**ARRAYCAT** (*expression, delimiter*)

Concatenates the values in a multiple valued variable or expression, to produce a single valued result. The first argument is a multiple valued expression, and the second argument is a string to be used as a delimiter between the values. An empty string may be specified as the delimiter. If any values of the first argument are null, they and their associated delimiters are skipped. The result has dimension 0 if the first argument has dimension 0, otherwise it has dimension 1.

**Example:** If the variable A has four values, “A1”, “A2”, Null, and “A4”, the expression ARRAYCAT (A, “-”) would produce a single string: “A1-A2-A4”.

**REPLACE** (*expression, expression, expression*)

Modifies character strings by replacing all occurrences of a sequence of characters with a replacement string. The first expression is the original character string to be modified. The second expression is a string to search for, and the third expression is a replacement string. Typically the second and third expressions will be single-valued, but the first expression can be multiple valued. The third expression can be an empty string, which causes all occurrences of the second string to be removed.

**Example:** If the variable A has the value, “This is a new test”, the expression REPLACE (A, “new”, “good”) would produce: “This is a good test”.

**DATEFORMAT** (*expression, format*)

Applies custom formatting to date/time data values. The first expression is a variable containing dates, as retrieved from the database. The format is a string in quotes that describes the desired date format. It follows the date format rules for the Microsoft .Net Framework, which is described in detail on their website. In general, there are two kinds of date formats: a single letter format specifies one of the standard formats, and a multiple-character string defines a custom format.

Standard formats are listed at <http://msdn.microsoft.com/en-us/library/az4se3k1.aspx>. Examples include:

“d”	Short date format	6/5/2009
“D”	Long date format	Friday, June 5, 2009
“g”	General date format (short time)	6/5/2009 1:45 PM
“G”	General date format (long time)	6/5/2009 1:45:30 PM

Custom formats are listed at <http://msdn.microsoft.com/en-us/library/8kb3ddd4.aspx>. Examples include:

“M/d/yy”	6/5/09
“MMMM d, yyyy”	June 5, 2009
“MM/dd/yyyy H:mm”	06/05/2009 13:45

**Example:** If the variable A has the value, “10/06/2008 14:10:05.1554”, the expression DATEFORMAT (A, “d”) would produce: “10/6/2008”.

**NAMECAP (expression, expression, type, expression)**

Applies standard capitalization rules for mapunit and component names. This is designed for use with the Calculations for mapunit and component name capitalization. The first parameter is an array of component names that are exceptions to the standard. Normally this will come from the “name\_exceptions” file distributed with NASIS. The second parameter is the array of component or mapunit names to be standardized, and the third is “C” for components or “M” for mapunits. The fourth parameter is the mapunit kind, and is only required for mapunits.

In general, the capitalization standard is that the first letter of each component name is capitalized and everything else is in lower case. The names in the exception list don’t follow this rule. In addition, mapunit names are arranged in a standard form depending on the mapunit kind.

**Example:**

```
INPUT exceptions FILE "name_exceptions".
DEFINE stdnames NAMECAP(exceptions, muname, "M", mukind).
```

**Function Expressions**

The following function expressions can use either character or numeric values, and produce results in the same type as the input, unless specified otherwise.

**NEW (*expression*)**

Returns True (1) if the value of the expression is different from the value it had in the previous cycle of the script, or False (0) if the value is the same.

**Example:** the expression NEW (mapunit\_symbol) will be True each time the mapunit symbol changes.

**CODENAME (*expression* [, *name* ])**

Returns the code name for the code value given by *expression*, using the data dictionary domain of the element *name*. The *name* must be a data element name or its alias from an EXEC SQL statement. The value of the expression must be a number representing the internal identifier for a code. This is the value normally returned by a query. If *expression* is the same as *name* you do not have to specify it twice.

**Example:** if the variable *compkind* were returned from a query, the expression CODENAME(compkind) would produce a string normally displayed in NASIS for that data element, such as “series”. Code names are generally in lower case. The expression CODENAME(val, compkind), where *val* is a variable from a

DEFINE statement, would produce the code name for a compkind whose value is in the variable *val*.

**CODELABEL (*expression* [ , *name* ] )**

Returns the code label for the code value given by *expression*, using the data dictionary domain of the element *name*. This operates just like CODENAME. The code label is typically the same as the code name but is capitalized properly for use in reports.

**Example:** in the above example, the expression CODELABEL(compkind) would produce “Series”.

**APPEND (*expression*, *expression* )**

Combines the values from two variables or expressions into a single variable. If the first expression has dimension *n* and the second expression has dimension *m*, the result of APPEND has dimension *n+m* and contains all the values from the first expression followed by the values from the second. If an expression has dimension 0, it does not add anything to the result.

**Example:** if the variable A has three values, 1, 2, and NULL, and the variable B has the value 3, the expression APPEND(A,B) would have four values: 1, 2, NULL, 3.

**ARRAYCOUNT (*expression*)**

Counts the number of non-null values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single numeric value of zero or more.

**Example:** if the variable A has three values, 1, 2, and NULL, the expression ARRAYCOUNT(A) would produce the result 2.

**ARRAYMIN (*expression*)**

Computes the minimum of the values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. In this case, a null value is not considered to be smaller than a non-null value. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has three values, 1, 2, and 3, the expression ARRAYMIN(A) would produce the result 1.

**ARRAYMAX (*expression*)**

Computes the maximum of the values in a multiple valued expression. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has three values, “X”, “Y”, and “Z”, the expression ARRAYMAX(A) would produce the result “Z”.

### **ARRAYMEDIAN** (*expression*)

Locates the median value in a multiple valued expression, by sorting the non-null values and selecting the middle one. It can operate on either a character or numeric argument, but there is a slight difference in operation between the two. When there is an even number of values there is not a single middle value, so with numeric data the median is the average of the two middle values, and with character data the median is the larger of the two. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has three values, “X”, “Y”, and “Z”, the expression ARRAYMEDIAN(A) would produce the result “Y”.

### **ARRAYMODE** (*expression*)

Finds the modal value in a multiple valued expression by counting the occurrences of each distinct value and returning the value that occurs most often. In case of a tie, the smallest value is returned. It can operate on either a character or numeric argument, and will return a single value of the same type as its argument. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has four values, 2, 3, 5, and 3, the expression ARRAYMODE(A) would produce the result 3.

### **ARRAYSHIFT** (*expression, expression*)

Shifts the values in the first argument, which is a multiple valued variable, by the number of positions specified in the second argument, which has a single value. If the second argument (call it “n”) is positive, the values are shifted “up”, so that the value that was in position 1 moves to position n+1, and so on until the last n values are discarded. The first n array positions are assigned a null value. If the second argument is negative, the values are shifted in the opposite direction. The result has the same data type and number of values as the first argument.

**Example:** if the variable A has three values, 1, 2, and 3, the expression `ARRAYSHIFT(A, -1)` would produce a result with three values, 2, 3, and Null.

### **ARRAYPOSITION** (*expression*)

Produces a new array of the same dimension as the argument and each position of the array having a sequential number starting with 1. This can be useful with the `LOOKUP` function to pick out a specific item from an array.

**Example:** if the variable A has three values (“x”, “y”, null) the expression `ARRAYPOSITION(A)` would produce a result with three values (1, 2, 3).

The following statement would find the third value (if there is one) in the array A:  
`DEFINE third LOOKUP(3, ARRAYPOSITION(A), A).`

### **ARRAYROT** (*expression, expression*)

Operates like `ARRAYSHIFT` but performs a rotation of the values in the first argument. Values shifted off one end of the array are moved onto the other end. If the number of positions shifted is greater than the number of values, the effect is to perform more than one rotation, or a rotation modulo the dimension.

**Example:** if the variable A has three values, 1, 2, and 3, the expression `ARRAYROT(A, -4)` would produce a result with three values, 2, 3, and 1.

### **LOOKUP** ([ *expression, ] expression, expression*)

Selects values from an array based on an index or condition. With three parameters, the first expression is the key, which must be a single value, and the second expression is the index array. The key and the index must have the same type of data. If the key value is found in the index array, the value from the corresponding array position in the third expression is returned, otherwise the result is null. With two parameters, the first expression is evaluated as an array of true or false values. If a value is true, the corresponding array position in the second expression is returned. The result will have the data type of the last expression.

There is actually a close relationship between the two forms of `LOOKUP`. These two expressions produce the same result: `LOOKUP(a,b,c)` and `LOOKUP(a==b,c)`. Use whichever form is easier to understand.

If there is more than one match or true value, the result has the values from all matching/true rows, so it is possible for the result to have more than one value. The last two expressions must be arrays of equal dimension. A common error is to mismatch the dimensions of these two expressions, due to differences in the way they are aggregated.

**Example:** The variable *max\_thickness* has a single number, the variable *horizon\_thickness* has 6 numbers, and the variable *ph\_r* has 6 numbers. The

expression LOOKUP (*max\_thickness*, *horizon\_thickness*, *ph\_r*) or LOOKUP (*horizon\_thickness*==*max\_thickness*, *ph\_r*) would return the value of *ph\_r* from the horizon whose *horizon\_thickness* value equals the value of *max\_thickness*.

**COUNT (*expression*)**

Maintains a running count of the occurrences of the expression. On each cycle of the script the value of the expression is tested for a null, and if it's not null the counter's value is increased by one.

**Example:** a variable defined with the value COUNT(*musym*) could be printed at the end of a report to show the number of mapunits read (because *musym* can't be null).

**MIN (*expression*)**

Finds the smallest value of the expression. On each cycle of the script, the value of the expression is compared to an internal counter, and replaces the counter's value if the expression is smaller. If a null value for the expression is encountered, the result of MIN becomes and remains null.

Internal counters for the MIN function cannot be reset.

**Example:** a variable defined with the value MIN(*elevation*) could be printed at the end of a report to show the minimum of elevation.

**MAX (*expression*)**

Finds the largest value of the expression. On each cycle of the script, the value of the expression is compared to an internal counter, and replaces the counter's value if the expression is greater. Null values are smaller than any non-null value, so the result is only null if all input values are null.

Internal counters for the MAX function cannot be reset.

**Example:** a variable defined with the value MAX(*elevation*) could be printed at the end of a report to show the maximum of elevation.

**PRINTF ("*format*", *expression* [ , *expression* ] ... )**

Formats one or more expression values into a character string using the C function *sprintf* (same as the Prelude *sprintf*). The first argument is a format specification, which must have a single value, and the remaining arguments are expressions whose values are to be formatted. If any of the expressions are multiple valued,

the result is also multiple valued, and its dimension is that of the expression with the largest dimension.

It is the user's responsibility to see that the number and type of the expressions correspond to the format, as there is no checking performed. Character data should use the %s formatting code, and numeric data should use the %f or %g formatting code.

Null values in the expressions produce an unusual result. The formatted value plus all characters of the format string up to the next % sign are skipped.

**Example:** The variable *name* has one character value, "Bob". The variable *position* has two numeric values, 10 and 12. The expression `SPRINTF ("%s:%.f", name, position)` will produce a result containing two character values, "Bob:10" and "Bob:12".

## **USER**

The user name from the data dictionary.

**Example:** if the person running NASIS has the login name "rose", the expression `USER` will return a single character value, "rose".

## **TODAY**

The current date in mm/dd/yyyy format.

**Example:** the result of the expression `TODAY` might be "07/20/1998".

## **Numeric Functions**

The following function expressions operate on numeric values, and produce numeric results. If the input values are character strings they are first converted to numbers.

### **ARRAYSUM (*expression*)**

Computes the sum of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are treated as zeroes. The result is null only if the array has no values. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable *A* has three values, 1, 2, and 3, the expression `ARRAYSUM(A)` would produce the result 6.

### **ARRAYAVG (*expression*)**

Computes the average of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are not counted in the average. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has three values, 1, 2, and 3, the expression ARRAYAVG(A) would produce the result 2.

### **ARRAYSTDEV** (*expression*)

Computes the standard deviation of the values in a multiple valued expression. It expects a numeric argument, and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null they are not included in the computation. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0, otherwise it has dimension 1.

**Example:** if the variable A has three values, 1, 2, and 3, the expression ARRAYSTDEV(A) would produce the result 1.

### **WTAVG** (*expression, expression*)

Computes the sum of the first expression's values after multiplying each by a weighting factor, taken from the corresponding value of the second expression, then divides the result by the sum of the weights. The two expressions must be arrays of the same dimension. Individual null values are ignored in computing the average. The result is null if all the individual values are null. The result has dimension 0 if the original expressions have dimension 0, otherwise it has dimension 1.

**Example:** The variable *comp\_pct\_r* has 3 values (40, 30, 20) and the variable *elev\_r* has three values (1000, 1200, 900). The expression WTAVG (elevation, comp\_pct\_r) would produce the value 1044.44, which is the average of the *elevation* values, weighted by the *comp\_pct* values, or  $(1000*40 + 1200*30 + 900*20) / (40 + 30 + 20)$ .

### **SUM** (*expression*)

Computes a running total of the value of the expression. On each cycle of the script, the value of the expression is added to an internal counter. The result of the function is the value of that counter at each cycle. If a null value for the expression is encountered, the result of **SUM** becomes and remains null.

Internal counters for the **SUM** function cannot be reset. If you want to compute subtotals, use the **ASSIGN** statement to add the value of the expression to a

defined variable rather than an internal counter. Then a conditional expression can be used to reset the variable's value to 0 at the correct time.

**Example:** a variable defined with the value SUM(acres) could be printed at the end of a report to show the total of acres.

### **AVERAGE (*expression*)**

Computes a running average of the value of the expression. On each cycle of the script, the value of the expression is added to an internal counter, and the result is divided by the number of values processed. If a null value for the expression is encountered, the result of AVERAGE becomes and remains null..

Internal counters for the AVERAGE function cannot be reset.

**Example:** a variable defined with the value AVERAGE(elev\_r) could be printed at the end of a report to show the average of elevation.

### **LOGN (*expression*)**

Computes the natural logarithm of the expression.

**Example:** the expression LOGN(10) produces the value 2.302585.

### **LOG10 (*expression*)**

Computes the base 10 logarithm of the expression.

**Example:** the expression LOG10(10) produces the value 1.

### **EXP (*expression*)**

Computes the exponential ( $e^x$ ) of the expression.

**Example:** the expression EXP(1) produces the value of  $e$ , 2.718282.

### **COS (*expression*)**

Computes the cosine of the expression interpreted as an angle in radians.

**Example:** the expression COS(0) produces the value 1.

### **SIN (*expression*)**

Computes the sine of the expression interpreted as an angle in radians.

**Example:** the expression SIN(0) produces the value 0.

**TAN (*expression*)**

Computes the tangent of the expression interpreted as an angle in radians.

**Example:** the expression TAN(0) produces the value 0.

**ACOS (*expression*)**

Computes the arccosine of the expression, returning an angle in radians.

**Example:** the expression ACOS(0) produces the value of  $\pi/2$ , 1.570796.

**ASIN (*expression*)**

Computes the arcsine of the expression, returning an angle in radians.

**Example:** the expression ASIN(1) produces the value of  $\pi/2$ , 1.570796.

**ATAN (*expression*)**

Computes the arctangent of the expression, returning an angle in radians.

**Example:** the expression ATAN(1) produces the value of  $\pi/4$ , 0.785398.

**ATAN2 (*expression, expression*)**

Computes the angular component  $\theta$  of the polar coordinates  $(r, \theta)$  that are equivalent to the rectangular coordinates  $(x, y)$  given by the two expressions. This is the same as ATAN( $y / x$ ).

**Example:** the expression ATAN2(5, 5) produces the value of  $\pi/2$ , 1.570796.

**SQRT (*expression*)**

Computes the square root of the expression. Returns a null value if the expression is negative.

**Example:** the expression SQRT(2) produces the value 1.414214.

**ABS (*expression*)**

Computes the absolute value of the expression.

**Example:** the expression ABS(-10) produces the value 10.

**POW** (*expression, expression*)

Computes the value of the first expression raised to the power of the second expression.

**Example:** the expression POW(2, 5) produces the value 32.

**MOD** (*expression, expression*)

Computes the remainder after dividing the first expression by the second expression.

**Example:** the expression MOD(5, 2) produces the value 1.

**ROUND** (*expression [, expression]* )

Rounds off the value of the first expression to the number of decimal places specified by the second expression. If the second expression is not used, it is assumed to be zero, which means round off to the nearest whole number. When the second expression is a positive number, it specifies the number of places to the right of the decimal point to be preserved. If negative, it means round to the specified number of places to the left of the decimal point, as illustrated in the examples.

**Examples:** ROUND (15.751, 1) produces 15.8  
ROUND (15.751) produces 16  
ROUND (15.751, -1) produces 20

**REGROUP Expression**

The REGROUP expression is used to perform secondary aggregation of data. It operates a little like the AGGREGATE option in a query and can be used to perform a second level of aggregation when dealing with a complex data structure. It uses two expressions, which must be arrays of the same dimension. In the expression “REGROUP array BY array ...” the second array (the “BY” array) is used as a key for grouping the values from the first array (the data array). The result is a new array whose dimension is the number of unique values in the “BY” array. The values in the result are aggregates derived from each group of rows in the data array that have the same key value.

The aggregation function determines how these aggregates are produced. The types of aggregation are the same as the query AGGREGATE option, except that NONE and UNIQUE are not applicable in REGROUP, because there can be only one value in each position of the result array. The valid aggregations types are:

SUM	Computes the sum of the values in each group.
AVERAGE	Computes the average of the values in each group.
FIRST	Select the value from the first row of the group.
LAST	Select the value from the last row of the group.
MIN	Selects the smallest of the values in each group.
MAX	Selects the largest of the values in each group.
LIST	Concatenates the values (converted to character strings if numeric) into a single string with a delimiter between each value. If a quoted string is specified after the word LIST, that string is the delimiter, otherwise a comma and space are placed between each value.

Some additional rules on the REGROUP expression are:

- The “BY” array does not have to be sorted. REGROUP will always collect together all data values for each unique key value. However, the choice of value for FIRST or LAST will be affected by the order of values in the data array.
- Nulls in the data array are ignored during aggregation except for FIRST and LAST, which preserve a null if it is the first or last value found. If all data values for some key value are null the corresponding result value will be null.
- A null in the “BY” array is a valid key value and will produce a corresponding value in the result, aggregating all null key values together.

**Example:** These examples use the arrays A and B as inputs:

<b>A</b>	<b>B</b>
George	4
Abe	4
Sue	5
Sam	8
Mary	8
William	8

The arrays C and D are produced by the statements:

DEFINE C REGROUP A BY B AGGREGATE FIRST.

DEFINE D REGROUP A BY B AGGREGATE LIST “-“.

<b>C</b>	<b>D</b>
George	George-Abe
Sue	Sue
Sam	Sam-Mary-William

## DERIVE

### Syntax:

DERIVE derive\_list USING property\_call .

derive\_list  $\Rightarrow$  variable [ FROM identifier ] [ , variable [ FROM identifier ] ] ...

property\_call  $\Rightarrow$  [ "site\_name" : ] "property\_name"

[ ( argument [ , argument ] ... ) ]

argument  $\Rightarrow$   $\left\{ \begin{array}{l} \text{variable} \\ \text{element} \\ \text{literal} \end{array} \right\}$

### Used In:

#### Report, Property, Calculation

### Example:

```
DERIVE thickness FROM layer_thickness
USING "NSSC_Pangaea": "LAYER THICKNESS" (0, bottom).
```

The **DERIVE** statement invokes a property script to produce values for one or more variables. Each name listed after the keyword **DERIVE** becomes a local variable in the script where it occurs. It is assigned the value of the variable in the property script whose name follows the keyword **FROM**. If the names before and after **FROM** are the same, the **FROM** phrase may be omitted. The property must have the same base table as the calling script, and the scripts are automatically synchronized to return values for the current row of the base table.

The name of the property must be in quotes, and must match the property name in the Property table exactly, including case and punctuation. The NASIS site name is optional, but should be placed before the property name to ensure that the name is unique.

A list of arguments must be given after the property name if the property script has an **ACCEPT** statement. The order of the arguments in the **DERIVE** statement must correspond to the order of the input variables in the **ACCEPT** statement.

The arguments can be input column names, variables, or numeric or character constants in the calling script. However, recall that **DERIVE** statements are always executed before **DEFINE** statements. If an argument is a variable which is computed in a **DEFINE** statement, its value will be whatever is left over from the previous script cycle, even if the **DEFINE** appears in the script before the **DERIVE**. For this reason, arguments for **DERIVE** should be from an **ACCEPT**, an **EXEC SQL**, or constants.

**EXEC SQL****Syntax:**

EXEC SQL sql-select [ sort\_specification ] [ aggregation ] .

sql-select  $\Rightarrow$  SELECT [ DISTINCT ] [ TOP n ] input\_column [ , input\_column ] ...

FROM table\_spec [ , table\_spec ] ...

[ WHERE where\_condition [ {AND | OR} where\_condition ] ... ]

[ SQL GROUP BY clause ] [ SQL HAVING clause ]

[ { SQL ORDER BY clause | SQL INTO TEMP clause } ] ;

input\_column  $\Rightarrow$  { element [ [AS] alias ]  
expression [AS] alias }

element  $\Rightarrow$   $\left[ \begin{array}{l} \{ TablePhysicalName \\ TableLogicalName \\ alias \} \end{array} \right] \cdot \left\{ \begin{array}{l} ColumnPhysicalName \\ ColumnLogicalName \end{array} \right\}$

alias  $\Rightarrow$  name

table\_spec  $\Rightarrow$   $\left[ \begin{array}{l} \{ EDIT \\ REAL \} \end{array} \right] [OUTER] \left\{ \begin{array}{l} tbl\_imp\_nm \\ tbl\_nm \end{array} \right\} [alias] [join]$

join  $\Rightarrow$   $\left[ \begin{array}{l} \{ INNER \\ \{ LEFT | RIGHT \} OUTER \\ CROSS \} \end{array} \right] JOIN table\_spec \left\{ \begin{array}{l} ON SQL WHERE condition \\ BY relationship\_name \end{array} \right\}$

where\_condition  $\Rightarrow$   $\left\{ \begin{array}{l} SQL WHERE condition \\ JOIN table TO table [ BY relationship\_name ] \end{array} \right\}$

**Used In:****Report, Property, Calculation****Example:**

```
EXEC SQL select areaname, legenddesc, musym, muname
from legend, lmapunit, mapunit, outer area
where join area to legend and join legend to lmapunit
and join lmapunit to mapunit;
```

An EXEC SQL statement defines a database query that supplies input to the report engine. Any database columns or expressions listed in the SELECT clause of the query may be used as variables in the rest of the script. A script almost always has a query, the exceptions being reports that get all their data from files, parameters or derived properties. The primary purpose of the EXEC SQL is to specify which data elements are necessary for the report.

The EXEC SQL statement is a variation of a standard SQL Select statement. It performs the same basic function, but has additional capabilities to make report writing easier. The additional capabilities include:

- Use of NASIS logical column names as well as database column names
- Simplified syntax to specify join conditions
- Extended sort types, such as case insensitive and symbol sort
- More powerful GROUP BY features in the AGGREGATE clause, including independent aggregation by column, and crosstab formatting.

A SQL Select statement begins with a SELECT clause. It has a list of database columns or expressions, following normal SQL syntax, and each column must have a unique name. If expressions are used in the select list, an alias must be used with the expression to provide a unique name. Besides allowing most standard SQL expressions, NASIS permits the functions CODENAME, CODELABEL, CODESEQ and CODEVAL with data elements that are stored as codes. These functions cause the query to return the name, label, sequence, or internal value for a code. If none of these functions is used the query returns the internal value.

The word DISTINCT following the word SELECT removes duplicate rows from the query results. If the combination of the values of all items listed in the SELECT clause is duplicated, only one occurrence will be produced. (In prior versions of NASIS the word UNIQUE could be used instead of DISTINCT, but this is no longer allowed in SQL).

The phrase TOP n following the word SELECT is a SQL feature that allows you to specify the maximum number of records to be returned from a query. The records are sorted on the columns specified in the ORDER BY clause, then up to "n" of them are used as report input. This is handy to use while testing a report.

The FROM clause specifies all the tables used in the query, and may specify aliases and joins. Table names used in a FROM clause must be defined in the NASIS data dictionary or in an INTO TEMP clause of a prior query. Aliases can also be used with table names, to provide a shorter name or to make the name unique.

A CVIR script is designed to search either the selected set or the full local database, depending on its type. Normally, reports search the selected set only, while calculations and properties search the whole local database. The keyword EDIT or REAL in the FROM clause is used to override the table search behavior on a table by table basis. If

the keyword **EDIT** is used only the selected set will be read, and if **REAL** is used the whole local database is read. If a report is run on the national database, the **EDIT** or **REAL** option is ignored and the whole national database is read.

The **FROM** clause can also contain specifications for joining tables using current SQL syntax (as well as the older syntax based on Informix). The newer style of join puts all the join conditions in the **FROM** clause, as in these examples:

```
FROM datamapunit inner join component by default
FROM datamapunit left outer join component on dmuiid=dmuiidref
```

For a complete discussion of this kind of join you'll have to find a SQL manual or class. Some of the key points are:

- When the join conditions begin with **ON**, standard SQL syntax applies. You must specify the exact columns to be matched in each of the tables. You can also add more conditions beyond just the key columns.
- When using the **BY** condition you specify a relationship name defined in the NASIS data dictionary. In most cases the relationship name is "default". If more than one relationship exists between a pair of tables you must use the correct name. The Info page for a table in NASIS will list the relationship names.

**Advanced Note:** There is an important case when this type of join must be used, and it's related to a difference between Informix and SQL Server. This is when an outer join is used and you need to apply additional selection criteria to the outer (not required) table. These criteria operate differently if they are in the **FROM** clause or the **WHERE** clause. Here are two examples:

A. `FROM component left outer join comonth by default  
and 'month' = 'jan'`

B. `FROM component left outer join comonth by default  
where 'month' = 'jan'`

Query A will produce a row for a Component that does not have a January in the Component Month table, but B will not. The reason is that the **WHERE** conditions are applied to the result of the join. In B, each Component will be joined up with its Component Month rows (if any), and then only those with January will be selected. In A, the selection of January records occurs during the join process. If there are none, the outer join applies and the Component is included in the output even though there is no matching Component Month.

If you don't use the above type of join, the **WHERE** clause may use the "JOIN table TO table" syntax as in NASIS 5. The two tables in a **JOIN** condition must have a relationship recorded in the data dictionary. In this form the **BY** phrase can be omitted as long as only one relationship exists between the two tables. You can also use the word **OUTER** in the **FROM** clause in combination with this type of join specification. NASIS will internally convert it to the new style join.

Subqueries are also allowed in the WHERE clause, following SQL syntax with the extensions just described. It is permissible to use a JOIN condition between a table listed in the main query and a table listed in the subquery, which is a convenient way to create a coordinated subquery. Refer to SQL references for more information about subqueries. This is an advanced query topic.

The expressions in the WHERE clause may use variables defined in a PARAMETER statement, an ACCEPT statement or a prior query. The CVIR engine will plug in the values of such variables at the time the query is executed. If the variable is preceded by a dollar sign, such as \$name, the SQL query becomes a parameterized query. A parameterized query does not use the automatic query coordination specified by the [BASE TABLE](#) statement. The selection is controlled by the parameter value instead. With this technique a report can have a mix of queries with different aggregation levels. Examples can be found in the Mapunit Description reports.

Following the WHERE clause, the GROUP BY, HAVING, and ORDER BY clauses can be used with the normal SQL syntax. The INTO TEMP clause may also be used (provided the ORDER BY is not used) to direct the results of the query into a temporary database table. Subsequent queries in the same script or in subreport scripts can read from the temporary table as if it were a normal NASIS table. The column names in the temporary table are the column names (or aliases) from the SELECT clause. A query with an INTO TEMP clause should not be the only query in a report because it does not return any data that the report can use.

NASIS tables and columns may be called by either the logical name or the physical name, but must use the same name wherever referenced. A column name can be used alone if it is unique, otherwise the table name must be given also. If an alias is used for an column in a SELECT clause, that alias must be used everywhere instead of the column name. If the column is modal, the suffix (such as \_l or \_h) must be included in the name. The value from each column is converted into either a character string or a floating point number for use in later calculations. Numeric data elements, such as Int, Decimal, and Float, and Code elements, are converted to floating point, and everything else, including dates, is converted to character strings.

A semicolon is required to end the SQL portion of the EXEC SQL statement. Optional sort and aggregation clauses may follow the semicolon, and the whole statement is ended with a period. If neither the sort nor aggregation is used, both a semicolon and a period are still required.

A script may contain more than one query, in order to collect data from different hierarchic paths in the database. These types of data often cannot be retrieved in a single query without creating undesirable cross products. By using separate queries and aggregating the results, the data can be “de-normalized” so that data from separate paths appear as if they were repeating groups in the base table.

## EXEC SQL: Sort Specification

### Syntax:

sort\_specification  $\Rightarrow$  SORT [BY] sort\_key [, sort\_key ] ...

sort\_key  $\Rightarrow$  name  $\left[ \begin{array}{l} \{ \text{ASC}[\text{ENDING}] \} \\ \{ \text{DESC}[\text{ENDING}] \} \end{array} \right] \left[ \begin{array}{l} \{ \text{LEX}[\text{ICAL}] \\ \text{SYM}[\text{BOL}] \\ \text{INSEN}[\text{SITIVE}] \} \end{array} \right]$

### Example:

```
EXEC SQL select areaname, legenddesc, musym, muname,
lmapunit.seqnum
from legend, lmapunit, mapunit, outer area
where join area to legend and join legend to lmapunit
and join lmapunit to mapunit;
SORT BY areaname, lmapunit.seqnum DESC, musym SYMBOL.
```

**SORT** is an optional clause that may be added to a query to direct the CVIR engine to sort the records. Either the **ORDER BY** (which causes the database engine to do the sorting) or the **SORT** may be used, and the **SORT** takes precedence. The **SORT** clause is slower but provides more options than **ORDER BY**. The sort key names in the **SORT** clause must be column or alias names used in the **SELECT** clause (column numbers are no longer allowed). The direction of sorting (ascending or descending) can be specified for each sort key, with the default being ascending. The type of sort can also be specified as lexical (like a dictionary), symbol (used for symbols containing both letters and numbers), or insensitive (ignore upper and lower case distinctions). The default sort type is the one specified in the data dictionary for the column. The sort order and type keywords may be abbreviated as shown.

The difference between **SORT** and **ORDER BY** is important when the **TOP n** condition is used in the **SELECT** clause. **ORDER BY**, since it is performed by the database engine, happens before the "top n" records are selected, and **SORT** happens after. It could even be useful to specify different columns in **ORDER BY** and **SORT**, because the first controls which records appear and the second controls the order in which they print.

**EXEC SQL: Aggregation Specification****Syntax:**

```

aggregation ⇒ AGGREGATE [ ROWS [ BY ] identifier [ , identifier ] ... ]
               [ COLUMN identifier [ aggregate_function ]
                 [ , identifier [ aggregate_function ] ] ... ]
               [ CROSSTAB [ BY ] identifier [ value_specification ]
                 [ LABELS "string" [ , "string" ] ... ]
                 CELLS identifier [ , identifier ] ... ] .

```

```

identifier ⇒ { element
              }
              { alias
              }

```

```

aggregate_function ⇒ { SUM
                      }
                     { AVERAGE
                      }
                     { FIRST
                      }
                     { LAST
                      }
                     { MIN
                      }
                     { MAX
                      }
                     { LIST [ "string" ]
                      }
                     { NONE
                      }
                     { UNIQUE
                      }
                     [ GLOBAL ]

```

```

value_specification ⇒ { VALUE[S] ( field_value [ , field_value ] ... )
                      }
                      { INTERVAL[S] ( field_value [ , field_value ] ... )
                      }

```

```

field_value ⇒ literal

```

**Example:**

```

EXEC SQL select musym, muname, areaname,
lmuaoverlap.areaovacres acres
from mapunit, lmapunit, lmuaoverlap, laoverlap, area
where join area to laoverlap
and join laoverlap to lmuaoverlap
and join lmapunit to lmuaoverlap
and join lmapunit to mapunit;
SORT BY musym SYMBOL, areaname
AGGREGATE ROWS BY musym
COLUMN muname UNIQUE, acres SUM
CROSSTAB areaname CELLS acres.

```

The aggregation clause specifies how the input records are to be grouped, and what to do with the data in each group. The first query in a report (the “primary” query) can use the **ROWS** option to control how its records will be grouped, but the remaining queries (the “secondary” queries) cannot have a **ROWS** option and instead use a simple global aggregation. This difference is significant in the following explanations.

A primary query without a **ROWS** option uses no aggregation, meaning that data records are used one at a time exactly as they come from the query. When the **ROWS** option is used, each unique combination of values in the **ROWS** columns starts a new cycle of report processing. The query must be sorted on the columns listed after **ROWS**, and it may produce more than one record for each combination of the **ROWS** columns. When there are multiple records in a group, the input values in each column are combined according to the aggregation rules, to produce single values or arrays that can be used in further calculations or report output.

The behavior of row aggregation is illustrated in the following example. Suppose a query includes specifications to **SORT BY musym AGGREGATE ROWS BY musym**. Rows with the same **musym** will define the report cycle. Each group of rows with the same **musym** is one cycle, thus this example has eight rows, but only three cycles.

musym	compname	comppct_r
12A	Hamerly	80
12A	Vallers	15
12A	Hamre	5
26B	Windsor	90
26B	Deerfield	10
130C	Dacono	85
130C	Satanta	10
130C	Altvan	5

Aggregation rules for each column can be specified after the keyword **COLUMN**. The default aggregation is **UNIQUE** for columns that have no aggregation specified. This means that when the value in a column is the same for every row in a report cycle, only one value is returned for that column. If more than one value occurs in a cycle, an array is formed to return values for the column. Each distinct, non-null value is placed in a separate position of the array. The number of positions in the array (the dimension) can vary from one cycle to the next and from one column to another within a cycle.

The aggregation function **NONE** is similar to **UNIQUE** except that it does not eliminate duplicate or null values. If there is more than one input row in a cycle, the value from

each row is placed in a separate array position. For each cycle, every column with **NONE** aggregation will have the same dimension, and the values will be in the order of the input records.

The other aggregation functions are used to reduce multiple values for a column to a single value. The aggregations have no effect when only one record occurs in a cycle. The types of aggregation are:

- SUM**            Computes the sum of the column's values.
- AVERAGE**    Computes the average of the column's values.
- FIRST**        Select the value from the first record of the group (useful only if the input is sorted on this column).
- LAST**         Select the value from the last record of the group.
- MIN**          Selects the smallest of the column's values.
- MAX**          Selects the largest of the column's values.
- LIST**         Concatenates the values (converted to character strings if numeric) into a single string with a delimiter between each value. If a quoted string is specified after the word **LIST**, that string is the delimiter, otherwise a comma and space are placed between each value.

Given the example above, suppose the query includes specifications to **AGGREGATE ROWS BY musym COLUMN compname LIST, comppct\_r SUM**. Since aggregation for **musym** is not specified, the default aggregation of **UNIQUE** will be applied to that column to produce the following results. Note that the values in each column have been reduced to a single value for each cycle.

musym	compname	comppct_r
12A	Hamerly, Vallers, Hamre	100
26B	Windsor, Deerfield	100
130C	Dacono, Satanta, Altvan	100

The keyword **GLOBAL** may be used after the aggregation type for a column. This causes that particular column to be aggregated over the entire set of input data, rather than one cycle. The values for that column remain constant for the whole report. One use for global aggregation is to find data for report headings. If the first input cycle is missing some data needed in a heading, a global aggregation can find the first occurrence, or all unique occurrences of the data before the report processing actually begins.

The aggregation of secondary queries is similar to global aggregation. No **ROWS** option is allowed, and the whole set of records that the query produces in each report cycle is aggregated together. Column aggregation rules can be specified for the columns of a secondary query if the default option of **UNIQUE** aggregation is not wanted.

A secondary query normally is automatically coordinated with the primary query via the Base Table. A hidden Where condition is applied to a secondary query so that it produces only the rows that match the current base table row. If the secondary query is “parameterized”, meaning that it has references to variables in **\$name** format, the automatic coordination is not used. Instead the parameter values are inserted into the query each cycle to control the selection of records.

## CROSSTABS

The *crosstab* is a special type of aggregation that assigns values to positions in an array based on the value of a controlling column. It requires a **CROSSTAB** column, and one or more **CELLS** columns. These columns become arrays, but their dimension is determined not by the number of input rows in a cycle, but by the number of values for the crosstab. This dimension is constant for the entire query. The crosstab values are defined by the **VALUES** list, the **INTERVALS** list, or by default. The default is to use all the unique values found in the input for the crosstab column.

When doing a crosstab, for each cycle of the input, the arrays of values for the **CELLS** columns are first set to nulls. Then, for each input record, the value in the **CROSSTAB** column is examined. If it is one of the values in the **VALUES** list or the default list, or if it falls within one of the ranges in the **INTERVALS** list, its position in the list is noted. For each of the columns in the **CELLS** list, the value from the input record is placed in that position of the column’s array.

Within a cycle, the value of the crosstab column may repeat. If so, only one value can be stored in an array position for a cell, so the cell’s aggregation function is applied. If a cell has no aggregation, a data row is returned for each unique value. In each such data row, all aggregated columns will have constant values. The operation of crosstab can be illustrated using the following example data:

musym	muname	areaname	acres
10A	Alpha loam, 0 to 3	X	100
10A	Alpha loam, 0 to 3	X	200
10A	Alpha loam, 0 to 3	Y	300
10A	Alpha loam, 0 to 3	Z	400

10A	Alpha loam, 0 to 3	Z	500
10B	Alpha loam, 3 to 6	X	600
10B	Alpha loam, 3 to 6	Y	700
10B	Alpha loam, 3 to 6	Y	800

This table shows a small sample of input data from the example query above. The first case shows the results of a crosstab without aggregation of the crosstab cells:

AGGREGATE ROWS musym COLUMN muname UNIQUE  
CROSSTAB areaname CELLS acres.

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500
10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

In this example the column “musym” controls row aggregation. Column “muname” has the **UNIQUE** aggregation, so it maintains the values that correspond to each value of “musym”. Notice that if “muname” does not repeat at the same frequency as “musym”, it will become an array.

The columns “areaname” and “acres” become arrays of three positions each, because the crosstab column, “areaname”, has three distinct values in the input sample. The values placed in “areaname” are constant, namely the column grouping values “X”, “Y”, and “Z”. The cell column, “acres”, contains the acreage values for the corresponding position of “areaname”. Because there are multiple acreage values for each area in this example, the result has two rows for each symbol.

By adding an aggregation function to the “acres” columns, the crosstab produces just one row for each cycle defined by the **ROWS** condition, as in the following example:

AGGREGATE ROWS musym COLUMN muname UNIQUE, acres SUM  
CROSSTAB BY areaname CELLS acres

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	300	300	900

---

10B	Alpha loam, 3 to 6	X	Y	Z	600	1500	
-----	--------------------	---	---	---	-----	------	--

When **INTERVALS** are used for a crosstab, the list of field values must be numbers, in an increasing order. The number of intervals is one more than the number of values. If the intervals are specified as: **CROSSTAB BY x INTERVALS (n1, n2, n3)**, the crosstab will place the cell data into one of 4 array positions based on the value of the variable **x**:

$x \leq n1$        $n1 < x \leq n2$        $n2 < x \leq n3$        $n3 < x$

The **LABELS** specification can specify column headings for a report, which would otherwise be the field values for the **CROSSTAB BY** column. See the discussion about array specifications and column specifications under the **SECTION** statement for more information about formatting and printing cross tabulated data.

**FONT**

**Syntax:**

```
FONT "font name".
```

**Used In:**

**Report**

**Example:**

```
FONT "Courier".
```

This statement has no function in NASIS 6 and is ignored. NASIS does not have control over the font used to display a text style report when it is opened in an application of the user's choice. In an application like Notepad it is recommended that you use a Courier font so that the output will look like it did in NASIS 5. For HTML style reports the font is controlled by the style sheet or by attributes of the HTML tags.

**HEADER and FOOTER****Syntax:**

```
HEADER [ INITIAL ]
        line-specification ...

END HEADER .

FOOTER [ FINAL ]
        line-specification ...

END FOOTER .
```

**Used In:****Report (text style only)****Example:**

```
HEADER
AT CENTER "Sample Report".
SKIP 2 LINES.
END HEADER.
```

Defines the headers and footers for the report. There are four types of header/footer statements, and a report may contain no more than one of each type. All are optional. The default for **HEADER** and **FOOTER** is to print nothing. The default for **HEADER INITIAL** or **FOOTER FINAL** is to print the **HEADER** or **FOOTER**, respectively.

The regular header and footer are printed at the top and bottom, respectively, of each report page. The initial header and final footer are printed only once, at the beginning and end of the report instead of the regular header and footer. At the end of the report, if there is not enough room for the final footer (which could happen if the final footer uses more lines than the regular footer), the regular footer is printed on the last page of data, then the final footer is printed on a separate page. Each header or footer contains one or more line specifications as defined below (except for **NEW PAGE** commands). The text of headers and footers is generated one time, at the beginning of report execution, and reprinted at the top of each page. Page numbers, if included in headers and footers, will be substituted correctly. Data from the database used in headers or footers will come from the first input record only.

## INPUT

### Syntax:

```
INPUT input-list FILE filename [ DELIMITER "string" ] [ sort-specification ]  
[ aggregation ] .
```

input-list ⇒ input-column [ , input-column ] ...

input-column ⇒ *name* [ CHARACTER | NUMERIC ] [ alias ]

filename ⇒ "string" [ / "string" ] ...

### Used In:

#### Report, Property, Calculation

The INPUT statement reads data from a file into CVIR variables. Each column name in the input-list (or alias if used) becomes a variable in the script. If the column name is a NASIS data element name the data type for the column is the same as the element's. If not, either CHARACTER or NUMERIC must be specified.

The file name is entered in quotes. The whole file path can be within one pair of quotes, or there can be several parts within quotes and separated by a slash. A full path name is required if you are using a file that you have created on your computer. If just the file name part is supplied the file must be in the NASIS installation under the "data\input files" folder. Examples are:

```
INPUT col1, col2 FILE "lookup.data".
```

```
INPUT areaname, areaacres FILE "C:\My Documents\datafile".
```

The first example uses just a file name, so it is presumed to be a file distributed with NASIS. The second example uses a file in the My Documents folder.

Input from a file can be aggregated, as described above for queries, to produce single or multiple valued variables. If the INPUT statement precedes any queries in a script, the report will have a cycle for each input record just as if a query were used. A BASE TABLE declaration cannot be used in this case. If the INPUT appears after a query, the aggregation for the INPUT is assumed to be global, similar to a parameterized query.

The input record must be in ASCII character format. A delimiter follows each data value in the input record. Any character string can be specified as the delimiter. The default is the "pipe" character, "|".

## INTERPRET

### Syntax:

```

INTERPRET rule [, rule] ...
    [ MAX REASONS max_value ]
    [ MAX RULEDEPTH max_value ]
    [ sort-specification ] [ aggregation ]

rule => [ "site_name" : ]"rule_name"

max_value => { number | variable }

```

### Used In:

#### Report

Generates interpretations for inclusion in a report. One or more rules can be specified, and the interpretation values will be computed during each cycle of the report. The interpretations are produced for the report's **BASE TABLE**. The same table must be used as the base table for the properties used in the interpretation.

Note that the process for generating interpretations is not the same as in NASIS 5. Interps were formerly written to a temporary database table and retrieved with a query. In NASIS 6 the interp generator works more like a secondary query. During each report cycle interpretation data is generated for a single row. If the base table is component, interps are generated for one component at a time. The results become report variables, as listed below. Sorting and aggregating are available on the set of interp results for each component.

The rule list specifies rules whose values will be generated. Each rule name is written as "Site Name": "Rule Name". Site Name is the NASIS site that owns the rule, and Rule Name is the name of the rule. Each name must be in quotes. This can be written as just "Rule Name" since rule names are unique. Any rule in the NASIS database can be used in a report. If a **PARAMETER** is defined with the option **ELEMENT rule.rulename**, the parameter name may be used as a rule list.

The optional phrase **MAX REASONS** can be used to limit the number of reasons (sub-rules) whose results will be returned from the interpretation. All sub-rules are used to derive the interpretation results; this only limits how many are returned in report variables. If the number of reasons is zero or this phrase is omitted, all sub-rule results will be included, even the insignificant ones. If the number is greater than zero, the highest *n* significant sub-rule results will be returned. Significant means non-zero values for limitation type interps, or values other than 1 for suitability type interps. The sub-rules are always sorted so the most significant values are first.

The optional phrase `MAX RULEDEPTH` can be used to limit the number of levels of subrules returned. A rule depth of 0 means use only the main interp rating and no reasons, depth 1 means the main interp and its first level of reasons. If this phrase is omitted, subrules to the maximum depth will be returned.

**Examples:**

```
INTERPRET "ENG - Shallow Excavations".
```

Generates the "Shallow Excavations" interpretation for each component selected by the report query, and returns its results along with all sub-rule results.

```
INTERPRET "NSSC_Pangaea": "FOR-Harvest Equipment Operability",
          "NSSC_Pangaea": "FOR-Log Landing Suitability"
          MAX REASONS 5.
```

Generates results for two interpretations and returns up to 5 sub-rule results for each rule. Only non-zero sub-rule results will be returned. Notice that the rule names are fully qualified by NASIS site name.

**Using Interpretations in Reports:**

The results of the `INTERPRET` command are placed in report variables and aggregated according to the aggregation rules for secondary queries (see sorting and aggregation beginning on page 37). Unless `MAX RULEDEPTH 0` is specified, the interp generator will produce more than one value in each variable. Usually you will need to specify the `NONE` aggregation type for each column you want to use in the report, since the default aggregation type is `UNIQUE`. Crosstab aggregation is also available for reports that use more than one interpretation.

The report variables produced by the `INTERPRET` statement are:

Variable name	Description
PrimaryRuleInterpRuleID	The rule id of the top level rule.
PrimaryRuleInterpRuleName	The name of the top level rule.
InterpRuleID	Rule id of the rule or subrule that produced the rating values.
InterpRuleName	Name of the rule or subrule.
InterpRuleDepth	An indicator of the depth of the rating, where 0 is the top level.
InterpRuleResultSequence	The sort sequence of the ratings within a top level rule.

RatingValueLowLow	The fuzzy value of the minimum rating for the rule or subrule.
RatingClassNameLowLow	The rating class name of the minimum rating.
RatingValueLowRV	The fuzzy value of the minimum of the representative values of the ratings.
RatingClassNameLowRV	The rating class name of the minimum of the representative values of the ratings.
RatingValueHighRV	The fuzzy value of the maximum of the representative values of the ratings.
RatingClassNameHighRV	The rating class name of the maximum of the representative values of the ratings.
RatingValueHighHigh	The fuzzy value of the maximum rating for the rule.
RatingClassNameHighHigh	The rating class name of the maximum rating.
NullPropertyData	True/false indicator that null data was produced by a Property
DefaultPropertyData	True/false indicator that default values were used to replace null values from a Property
InconsistentPropertyData	True/false indicator that inconsistent data were detected from a Property.

Rating values can be printed either as fuzzy values (numbers between 0 and 1) or as rating class names, or both. The values of the interp variables are sorted on `InterpRuleResultSequence`, which is a number assigned by the interpretation engine such that each subrule will come out after its parent rule, with the most significant rating values first. `InterpRuleDepth` can be used with the **NEST** option (described in column layout specifications under the **SECTION** statement) to print subrules indented below their parent rules.

**MARGIN****Syntax:**

```
MARGIN [ LEFT number [IN] ] [ RIGHT number [IN] ]  
[ TOP number [IN] ] [ BOTTOM number [IN] ] .
```

**Used In:****Report (text style only)****Example:**

```
MARGIN TOP 1 inch BOTTOM 1 inch.
```

Defines margins for the pages of text style reports. Defaults are one half inch for all margins. If margins are specified with IN, INCH, or INCHES, they are measured in inches, otherwise they are in lines or characters. The relationship between lines, characters and inches is defined by the PITCH specification.

The margin specifications are used to determine how much text can be placed on a page of output. However, the text file produced by NASIS does not have blank lines for top and bottom margins, nor blank spaces for the left and right margins. The user is responsible for setting appropriate margins in the application used to display the report.

If the page length is UNLIMITED the top and bottom margins are ignored, and if the page width is UNLIMITED the left and right margins are ignored.

**PAGE****Syntax:**

```
PAGE [ LENGTH { number [IN] | UNLIMITED } ]  
      [ WIDTH { number [IN] | UNLIMITED } ] .  
  
PAGE PAD  
      line-specification ...  
  
END PAGE PAD .
```

**Used In:****Report (text style only)****Example:**

```
PAGE WIDTH 144 LENGTH 88.  
  
PAGE PAD  
  USING normal_template.  
END PAGE PAD.
```

The Page Length/Width statement defines the size of the assumed page for report output. The default size is length 11 inches and width 8.5 inches. If sizes are specified with IN, INCH, or INCHES, they are measured in inches, otherwise they are in lines or characters. The relationship between lines, characters and inches is defined by the PITCH specification.

NASIS does not control the final appearance of the report, since it is displayed in an application of the user's choosing. The user may have to set the page size, margin and font in that application in order to produce printed output that matches the desired pagination.

Length can be specified as UNLIMITED, which means that the whole report is treated as a single page. This can be used in reports that are for screen display or for saving as text, but if the output is sent to a printer there will be no headings on each page.

Width can also be specified as UNLIMITED, which means that report lines are as long as the data in them requires. This is useful when the report output is saved as text, but sending the output to a printer would probably not be desirable.

PAGE PAD is used when lines to fill any unused space at the end of a page, or when the FILL command is used. The line specification in the PAGE PAD block is printed instead of the default padding, which is a blank line. If the block contains more than one line, the whole block of lines is printed repeatedly to fill the required space.

**PARAMETER**

**Syntax:**

PARAMETER name [ parameter\_attribute ] ... .

parameter\_attribute ⇒ {  
 ELEMENT element  
 PROMPT "*string*"  
 MULTIPLE  
 REQUIRED  
 MAX n  
 SEARCH  
 SELECTED  
 value\_type

value\_type ⇒ {  
 CHARACTER | CHAR  
 NUMERIC | NUM  
 BOOLEAN | BOOL  
 CODEVAL  
 CODESEQ  
 CODENAME  
 OBJECT  
 OBJECTID

**Used In:**

**Report**

A PARAMETER allows the user to customize the report script through a dialog. Parameter names are variables in the report script similar to variables created by the DEFINE statement, and may have zero or more values. Parameters are commonly used in WHERE clauses, or to provide a rule name in the INTERPRET statement, or for column names in a CROSSTAB.

The PARAMETER definition statement is normally placed at the beginning of the report script. For compatibility with older report scripts, the statement may begin with a # symbol like a comment, but the # is no longer required. Following the parameter name, one or more attributes may be specified to customize the parameter dialog. The attributes are:

ELEMENT means that the parameter takes the same values as the named data element. If the element has a choice list, that list appears in the parameter dialog.

The element's data type will also apply to the parameter value(s). The element name should be written as *tbl\_nm.elm\_nm* to be sure that the name is unique.

**PROMPT** provides a label for the input field in the parameter dialog. This can give the user hints on how to fill in the parameter value. If no prompt is provided, the Label field from the **ELEMENT** definition is used as the prompt. If neither **PROMPT** nor **ELEMENT** is provided, the parameter name is used as the prompt.

**MULTIPLE** means that more than one value can be entered for the parameter. If a choice list is used, multiple choices can be selected. The result of the parameter dialog is a multiple-valued variable.

**REQUIRED** means that at least one value must be entered for the parameter. This option is useful if the parameter is intended to be used in a query to select data, and you want to make sure that something is selected.

**MAX n** implies **MULTIPLE** and also puts an upper limit on the number of values that can be selected.

**SEARCH** means that a choice list will be built for use in the parameter dialog by searching the database for all unique values entered for the specified data element. The **ELEMENT** attribute must be used with **SEARCH**. Note that it could take some time to build a choice list if the element is in a table with many rows.

**SELECTED** is like **SEARCH** but only searches the data in the current selected set. This will present the user with a list of choices that could actually appear in the report. An example is a choice list for crop name. Normally this would list all crop names in the domain, but with **SELECTED** the choice list only includes crops that actually occur in the selected set.

The value type option allows the type of the parameter to be specified when the **ELEMENT** attribute is not used, or when code conversions are needed. The type tells the parameter dialog how to format the parameter value. Only one value type may be specified. The allowed types are:

**CHARACTER** means that the value is composed of any printable characters. This is the default if neither type nor **ELEMENT** is specified.

**NUMERIC** means that the user must enter a number.

**BOOLEAN** means that the parameter dialog will display a toggle button instead of a data entry field. The parameter's value is numeric, and contains a 1 or zero indicating whether or not the user toggled the button.

**CODEVAL** can be used when the parameter refers to a data element that uses codes. This option specifies that the parameter value will be returned as the

code's value, although the choice list contains code names. The default for coded elements is **CODEVAL**.

**CODESEQ** can be used when the parameter refers to a data element that uses codes. The parameter value will be returned as the code's sequence number.

**CODENAME** can be used when the parameter refers to a data element that uses codes. The parameter value will be returned as the code's name.

**OBJECT** means that the parameter is an object name, such as a rule or property name. The parameter dialog displays a choice list for selecting NASIS site and object names. The parameter must refer to an element in the root table of a NASIS object, typically the name column. The value returned is in the format used in the **DERIVE** and **INTERPRET** statements, namely "site": "name". If used with the **MULTIPLE** option, the user can select multiple object names.

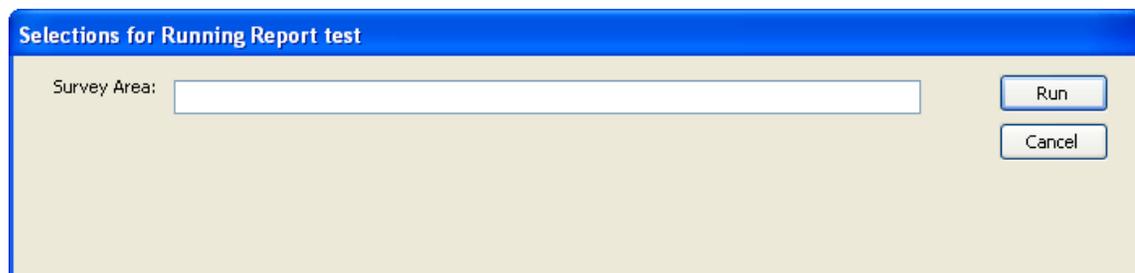
**OBJECTID** produces the same kind of choice list as **OBJECT**, but the value returned is the record ID of the selected item(s). This option is used when you want to query for a specific item or items, as opposed to the **OBJECT** option which returns a value that is not usable in a query Where clause.

The following fragments of report scripts illustrate the use of parameters:

```
PARAMETER aname ELEMENT area.areaname PROMPT "Survey Area".
```

```
EXEC SQL select ... where area.areaname = aname and ...
```

This asks the user to provide a survey area name, which is then used in a query to get records for the selected area. The parameter dialog would look like:



The image shows a dialog box with a blue title bar that reads "Selections for Running Report test". Inside the dialog, there is a label "Survey Area:" followed by a white text input field. To the right of the input field are two buttons: "Run" and "Cancel".

```
PARAMETER crops ELEMENT dmucropyld.cropname MULTIPLE SELECTED.
```

```
...
```

```
... CROSSTAB BY dmucropyld.cropname VALUES crops
```

This example allows the user to select one or more crop names from a choice list based on the contents of the selected set. The names will be used as column headings in a crop yield report. The prompt will be the label for the element *dmucropyld.cropname*, which is "Crop Name", as shown:

**Selections for Running Report test**

Crop Name:

- alfalfa hay
- barley
- beans, dry pinto
- corn silage
- oats
- pasture
- wheat
- wheat, winter

Run

Cancel

## PITCH

### Syntax:

PITCH [ HORIZONTAL *number* ] [ VERTICAL *number* ] .

Defines the character spacing, in characters or lines per inch. The default is horizontal 10 characters per inch and vertical 6 lines per inch, which corresponds to a 12-point fixed-width font such as Courier. The pitch specifications are used in combination with the page width, length and margins specified in the PAGE statement to determine how much text will fit on a page of output.

### Used In:

**Report (text style only)**

### Example:

```
PITCH HORIZONTAL 17 VERTICAL 8.
```

**SECTION****Syntax:**

```
SECTION [ section-name ] [ keep-option ] [ condition ]
[ HEADING output-specification ... ]
[ DATA output-specification ... ]
```

```
END SECTION .
```

section-name ⇒ *name*

**Used In:****Report****Example:**

```
SECTION WHEN LAST OF musym KEEP WITH main
DATA
  AT 40 "-----".
  AT 40 total_acres width 10 decimal 2.
END SECTION.
```

A report section defines a block of report output that is produced as a unit. A section can be unconditional, meaning that the section's data block is printed on each cycle of the report's main query, or it can be printed only when certain conditions occur. A report can have any number of sections. The sections are printed in the order determined by their conditions, as discussed below under Section Conditions.

For a simple example, imagine a report script having a section "A", which prints the mapunit symbol and mapunit name, followed by a section "B" that prints the component name. Section B is unconditional, and section A prints whenever the value of the variable "musym" changes. This would be defined in the following manner:

```
SECTION A WHEN FIRST OF musym
DATA
  AT LEFT musym, muname.
END SECTION.
```

```
SECTION B
DATA
  AT LEFT compname.
END SECTION.
```

The output of the report might be:

```
12A Hamerly-Vallers complex, 0 to 2 percent slopes
Hamerly
Vallers
```

Hamre  
26A Windsor loamy sand, 0 to 3 percent slopes  
Windsor

This would be produced from a query returning 4 records for the two mapunits. The first mapunit has three components and the second mapunit has one component. Since section A has a “first of” condition, it is printed before unconditional sections when a new value of *musym* is encountered. Then section B is printed for each input record until a change in *musym* occurs. Then section A is printed again, and finally section B is printed for the last record.

To define a section, specify one or more of the following features, each of which is discussed in more detail later. Note that in XML output style there is no concept of a “page”, so page layout features, such as **KEEP** and **HEADING**, are ignored.

1. A section can be given a name. Names are used in the **KEEP** option, and can be useful as documentation.
2. A **KEEP** controls the splitting of the section when the end of a page is reached.
3. A condition specifies when the section is used. If no condition is provided, the section appears for each report cycle.
4. If a **HEADING** block is provided, it prints at the top of the report page after the general header. If the section has no condition, the heading prints on every page, but if the section has a condition the heading only prints if the condition is true when it is time to start a page. The heading block contains one or more output specifications. If any data element values are printed in a heading, they will come from the record being processed at the time the heading prints (Note that this differs from the use of data in headers and footers).
5. If a **DATA** block is provided, it prints on each report cycle for which the condition holds. The data block contains one or more output specifications, and each of them can have an **IF** condition attached. The actual number of lines of output produced by a section is determined by many factors, including the conditions, the number of values stored in the variables being printed, and the length of text fields.

**SECTION: Conditions****Syntax:**

$$\text{condition} \Rightarrow \text{WHEN} \left\{ \begin{array}{l} \text{boolean\_expression} \\ \text{break\_condition} \\ \text{AT START} \\ \text{AT END} \\ \text{NO DATA} \end{array} \right\}$$

$$\text{break\_condition} \Rightarrow \left\{ \begin{array}{l} \text{FIRST} \\ \text{LAST} \end{array} \right\} [ \text{OF} ] \text{ identifier } [ , \text{ identifier} ]$$
**Example:**

```
SECTION WHEN type == 2
```

A condition can be an ordinary Boolean expression based on data from the database or report variables. In this case, the section prints whenever the condition evaluates to True. Boolean expressions are described under the **DEFINE** statement.

Another form of the condition detects control breaks in the report data. This type of condition begins with the keyword **FIRST** or **LAST**. At least one of the identifiers in the break condition should be a data element in the sort key for the main report query. A control break occurs when the value of any specified element, or of any element higher in the sort key, changes. The choice of **FIRST** or **LAST** in the break condition determines which data are used for the lines printed in the section. With **FIRST**, the first record with the new value of the control variable is used, while **LAST** uses the last record with the old value. The **LAST** condition would be used for printing subtotals for a group of records, while **FIRST** would be used for printing a heading line before a group of records.

The remaining conditions are used for special conditions that occur no more than once in a report.

The **AT START** condition means that the section prints before any other sections (but after the headers), while an **AT END** section prints after the last data record (but before the footers). The default for these sections is no printing.

A **NO DATA** section prints only if there are no input records, and could be used print a message such as “No data found”. If the **NO DATA** section is not used and there is no input, no report output is produced. Instead, a warning dialog is displayed to the user.

When a heading block is specified in an unconditional section, the result is easy to visualize; the heading lines print on each report page following the page header. The

headings appear in the order that the sections are defined. To reduce confusion, it is a good idea to include all unconditional headings in a single section, and place this section first in the script. In a simple report, both headings and data can be specified in the same unconditional section.

The operation of headings in conditional sections can be a little unexpected. These headings only print if a page break occurs while the conditional section is being printed. It helps to arrange for a page break to occur just before printing the conditional section. This feature can require some trial and error to get the desired results.

Heading lines can contain references to data elements or variables, whose values print in the heading. Note that headings are generated each time a new page begins, so the heading will contain values in effect at the time they print. In particular, a **LAST OF** section will use values from the last record before the control break, and a **FIRST OF** section will use values from the new record (the one causing the control break). Note however, if a **LAST OF** section (or any other type of section) causes a page break, all the headings on the new page will use data from the new record.

The order of processing for section conditions is:

1. **AT START** (only once per report)
2. **FIRST OF** (per report cycle)
3. Other sections, in the order they appear in the script
4. **LAST OF** (per report cycle)
5. **AT END** (only once per report)

**SECTION: KEEP option****Syntax:**

keep-option ⇒ { NO KEEP | KEEP WITH section-name [, section\_name ] ... }

**Example:**

```
SECTION b KEEP WITH A
```

The **KEEP** option controls what happens when the end of a page is reached while a section is being printed. Without any **KEEP** option, the default behavior is to allow a page break to occur after printing all the lines defined for one section occurrence. If the **DATA** block contains more than one line specification, or if continuation lines are needed for long text fields, these output lines will be kept together on a page. The **NO KEEP** overrides this by allowing page breaks between lines of a section, although text continuation is still kept on a page if possible.

The **KEEP WITH** option specifies other sections with which this section is linked. This means that when a section immediately follows an occurrence of one of its “keep with” sections, the data block for the new section occurrence must fit on the same page as the last line of data in the “keep with” section. If there is not room, a page break is inserted before the last keep block of the named section.

**KEEP** options are ignored when XML style output is produced. All page layout is controlled by the style sheet applied to the out put of the report generator.

**SECTION: Output Specifications****Syntax:**

output-specification  $\Rightarrow$  [ IF expression ] line\_content

line\_content  $\Rightarrow$  {  
 SKIP *number* {LINES|INCHES}.  
 FILL *number* {LINES|INCHES}.  
 NEW PAGE .  
 INCLUDE subreport [ ( argument [, argument ]... )  
 USING template\_name column\_spec [, column\_spec]...  
 at\_statement  
 element\_statement  
 }

subreport  $\Rightarrow$  [ "*site\_name*" : ] "*report\_name*"

argument  $\Rightarrow$  {  
 variable  
 element  
 literal  
 }

**Examples:**

```
SKIP 2 LINES.
AT LEFT musym WIDTH 8, muname WIDTH 50.
IF comp_pct > 10 USING comp_tmpl compname, slope_l, slope_h.
ELEMENT "tr" musym TAG "td", muname TAG "td".
INCLUDE "MLRA10_Office":"Flood Subreport" (dmudbsidref, coiid).
```

An output specification is used either to control spacing on the page or to produce actual report output. Output specifications can be either conditional or unconditional. When the IF clause is used, the IF expression is evaluated each time the section is processed. The expression follows the same rules as expressions for the DEFINE statement (see page 17). If it results in a True (non-zero) value, the output content is produced. If the value of the expression is False (a null, a zero or an empty character string) nothing is output. Without the IF clause, the output is always produced when its section is printed.

The output content is sometimes called a "logical line" because it is a single unit of output, even though it may include several "physical" lines on the report page. For example, if a logical line contains a text field it may require several lines on the page to print all the text. Depending on the KEEP rules a whole logical line is normally kept together on one page, unless the text requires more than a full page to print.

The line\_content portion of the command describes the output:

1. The **SKIP** command produces the specified amount of blank space. Either **LINES** or **INCHES** must be specified for the amount to be skipped. When page formatting is in effect, skip lines are not carried over past the bottom of a page. **SKIP** has no effect in XML style output.
2. The **FILL** command is like the **SKIP** command, but it fills the specified space with repetitions of the **PAGE PAD** block.
3. The **NEW PAGE** fills out the page with repetitions of the **PAGE PAD**, then prints the footer, starts a new page, and prints the header. If a **NEW PAGE** occurs at the very end of a report, the report generator will ignore it and not print an extra blank page. **NEW PAGE** has no effect in XML style output.
4. The **INCLUDE** command runs another report and inserts its output as a logical line in the first report. Parameters may be passed to the subreport, and they must correspond with variables in the subreport's **ACCEPT** statement. Typically a record key would be passed as a parameter, which would be used by the subreport to query for information related to that record. See [Using Subreports](#) for more detail.
5. The **USING** statement specifies a template to serve as a format for the output. The column specifications in the **USING** statement are matched to the **FIELD** keywords in the template. The element or variable specified in the column spec is printed with the formatting defined in the template. But any formatting options specified in the **USING** override the corresponding options in the template. If the **USING** does not have as many columns as there are **FIELDs** in the template, the remaining fields are printed as blank. The **USING** may not have more columns than the template has **FIELDs**. Columns in the **USING** statement may not use the **ARRAY** or **FIELD** options.
6. A text style output line is created with the **AT** statement, as described below
7. XML style output is created with the **ELEMENT** statement.

## AT Statement

### Syntax:

at\_statement  $\Rightarrow$  AT position [ alignment ] column\_spec [, column\_spec ] ...

[ ; AT position [ alignment ] column\_spec [, column\_spec ] ... ] ... .

position  $\Rightarrow$  { *number* [ IN ] | LEFT | RIGHT | CENTER }

alignment  $\Rightarrow$  { TOP | BOTTOM | SAME }

### Examples:

```
AT LEFT musym WIDTH 8, muname WIDTH 50.  
AT CENTER title WIDTH 20 CENTERED; AT RIGHT date WIDTH 12.
```

The AT statement specifies one or more groups of columns to be placed at specific positions in an output line. An AT position can be a number, expressed in characters or inches from the left margin, or it can be the left, right, or center of the line, relative to the margins. The position may be followed by an alignment option, which defines where this group of columns appears vertically on the page relative to the previous AT group, as shown in the examples below. If no alignment is specified the default is TOP.

Following the position and alignment, one or more columns are specified. These columns print adjacent to each other in order from left to right, with each occupying the number of characters specified by its WIDTH. When a column has WIDTH UNLIMITED it uses as many characters as needed to output the data, which may vary from line to line.

If the columns are not supposed to be adjacent a new AT keyword and position may be used. The list of columns following the AT will begin at the new position. A semicolon must separate AT groups, as shown in the syntax. Note that when more than one column spec follows an AT RIGHT or AT CENTER, the group of columns is first strung together, then right justified or centered as a unit.

Another application for an AT group occurs when printing data from array variables. Within an AT group that contains array variables, corresponding values in each array variable will always come out on the same line. If some value in an array is text that wraps around to a new line blanks will be inserted in the other columns as needed to maintain alignment across columns. For columns in different AT groups there is no alignment across columns. Text wrapping can cause data from different array positions to appear on the same line (which is desirable in some reports). The following example illustrates this:

This line specification uses columns “name”, “age” and “score” from a query with aggregation type NONE, so each column contains an array of values. The column “text” comes from a different query and has only one value, which is a long text string.

AT LEFT name width 12, age width 6, score width 7; AT 28 text width 28.

This might produce the following output. Because it is in a different AT group, “text” wraps across several lines, and is not associated with any one of the name lines. But when a name wraps, the associated data stays in alignment.

Jones	30	5.9	This group of people has
Abercrombie-	52	5.4	responded to all of the
Fitch			surveys conducted since May,
Smith	27	6.1	1983.
Martinez	41	5.7	

---

The line with the name “Abercrombie-Fitch” requires two lines of output because the name doesn’t fit in 12 characters. The age and score are printed on the first of these lines, and a blank appears beneath them due to wrapping in the first column.

In some cases this might not be quite the desired output. If you want the age and score to appear lined up with the end of the name rather than the beginning, it would require use of the alignment option. Age and score would have to be in a separate AT group using **BOTTOM** alignment, meaning that they line up with the bottom of the previous AT group. The following example shows this:

AT LEFT name width 12; AT 13 BOTTOM age width 6, score width 7; AT 28 text width 28.

This version would produce the following output.

Jones	30	5.9	This group of people has
Abercrombie-			responded to all of the
Fitch	52	5.4	surveys conducted since May,
Smith	27	6.1	1983.
Martinez	41	5.7	

---

There is a third possible alignment option, **SAME**. This is used in cases where there are three or more AT groups, the second one has **BOTTOM** alignment, and both of the first two groups could have wrapping of long text. Then there are three possible places for the third AT group to line up: the original top line of the first group, the bottom of the first group (which is the same as the second group) or the bottom of the second group. These three positions correspond to the alignments **TOP**, **SAME**, and **BOTTOM**. In NASIS the national manuscript report Table E2 uses this feature, if you want to see an example.

**ELEMENT Statement****Syntax:**

```
output-specification ⇒ ELEMENT [ OPEN | CLOSE ] element-name
    [ attribute [ attribute ] ... ] [ value-tag ] [ column-spec [ , column-spec ] ... ]
element-name ⇒ "string"
attribute ⇒ ( "string", { "string" | variable } )
value-tag ⇒ VALUETAG "string" [ attribute [ attribute ] ... ]
```

**Examples:**

```
ELEMENT "para" ATTRIB ("role", "subhead") musym, ": ", muname.
ELEMENT "tr" musym TAG "td", muname TAG "td" ATTRIB ("role"
"namecol").
```

The **ELEMENT** statement creates output in Extensible Markup Language (XML) format. XML is an industry standard for exchanging information on the web. An [Introduction to XML](#) is available from W3C, the web standards organization.

HTML, the standard language for web pages, is a subset of XML and can be produced with the **ELEMENT** statement. There are many books on HTML, and a simple online reference is available at <http://www.htmlhelp.com/reference/html40>. When HTML is specified as the output format for a report, NASIS converts the XML generated by **ELEMENT** statements into HTML so that the output can be viewed directly in a browser. The conversion works best if the report follows the conventions documented in Appendix 1, which are based on a documentation standard called DocBook.

The **ELEMENT** statement includes an element name which appears in the opening and closing tags that surround the content of the element. The first example above uses the element name "para", which is the DocBook tag for a paragraph, and a class attribute of "subhead". The content of the element is three items, a value of musym, a colon-space, and a value of muname. The output generated by this element might look like this:

```
<para role="subhead">12: Alpha silt loam, 5 to 8 percent
slopes</para>
```

Any attributes applicable to an element can be added with the **ATTRIB** option. Within the parentheses, you provide an attribute name and the value. In this example, the attribute "role" is a standard DocBook attribute that links to a style sheet where the formatting for elements of role "subhead" is defined. An element statement may contain several attributes.

Because XML describes structure as well as data, there will commonly be a need to produce elements within elements. The CVIR language provides some ways to do this. The first is the **TAG** attribute, illustrated in the second example above. When **TAG** is used in a column-spec, XML tags are produced around that column's data. Following the **TAG**, additional **ATTRIB** options can be specified, and those attributes will go in the column's tag, rather than the outer element tag. The output might look like this:

```
<tr><td>12</td><td role="namecol">Alpha silt loam, 5 to 8 percent slopes</td></tr>
```

This is a typical DocBook specification for a row of a table (abbreviated "tr") containing two table data ("td") columns. A more common use of this would be to put the element definition in a **TEMPLATE** statement, then specify the variables to be output with a **USING** statement, as in this example which produces the same output:

```
TEMPLATE row1 ELEMENT "tr" FIELD TAG "td", FIELD TAG "td" ATTRIB ("role" "namecol").
```

```
USING row1 musym, muname.
```

A further level of XML structure can be applied when the variables being printed in the **ELEMENT** statement are arrays. The **VALUETAG** option is similar to the **TAG** option except that it places a tag around each value of the array, while the **TAG** would place the tag around the whole set of values. You can use both **TAG** and **VALUETAG** to get a nested tag effect. This example uses the **hzname** variable that has multiple values:

```
TEMPLATE row2 ELEMENT "tr" FIELD TAG "td", FIELD TAG "td" VALUETAG "para" ATTRIB ("role" "namecol").
```

```
USING row2 compname, hzname.
```

This puts each horizon name within a "para" tag so that it will appear on a separate line, and uses the "namecol" role to get the right formatting for that column. The output would be like this (indentation has been added to make it more readable):

```
<tr>
  <td>Alpha</td>
  <td>
    <para role="namecol">A</para>
    <para role="namecol">B</para>
    <para role="namecol">C</para>
  </td>
</tr>
```

This example has the **VALUETAG** nested within the **TAG** option. It is also possible to nest **TAG** options within a **VALUETAG**. In that case the outer tag is repeated for each set of inner tags, as in the following example.

```
TEMPLATE row3 ELEMENT "table" VALUETAG "tr" FIELD TAG "td", FIELD  
TAG "td" ATTRIB ("role", "number").
```

```
USING row3 hzname, hzdept_r.
```

This example produces a complete table displaying a set of horizon names and depths, which are assumed to be aggregated into multiple-valued variables. Each pair of values for hzname and hzdept\_r becomes a table row enclosed in a “tr” tag. The output might be:

```
<table>  
  <tr>  
    <td>A</td>  
    <td class="number">0</td>  
  </tr>  
  <tr>  
    <td>B</td>  
    <td class="number">15</td>  
  </tr>  
  <tr>  
    <td>C</td>  
    <td class="number">45</td>  
  </tr>  
</table>
```

This type of output works best if all the variables have the same number of values. If not, the rows don’t all have the same number of columns, which produces poor looking output when converted to an HTML page.

The examples so far have shown how to produce XML tags nested up the three levels deep, using **ELEMENT**, **TAG** and **VALUETAG**. Often it is necessary to add even more levels for larger structures. The last example showed a very simple table, but often a table will have too much information to fit in one **ELEMENT** statement. Also, a `<table>` element would be inside the main `<section>` element of a typical DocBook document. These larger structures usually encompass most if not all the data in a report, so the opening and closing tags cannot be produced in a single statement.

For more complex structures we use conditional sections together with the **ELEMENT OPEN** and **ELEMENT CLOSE** forms of the statement. **ELEMENT OPEN** produces only the opening tag, and it might be placed in a section with a condition of **WHEN AT START** or **WHEN FIRST OF**. The **ELEMENT CLOSE** statement produces the corresponding closing tag, and it would be in a section with condition **WHEN LAST OF** or **WHEN AT END**. Here is an example of a part of a script to produce an DocBook table:

```
SECTION WHEN AT START  
  DATA  
    ELEMENT OPEN "section" ATTRIB ("label", "SoilReport").  
    ELEMENT "title" reporttitle.
```

```
ELEMENT OPEN "table.  
ELEMENT OPEN "thead.  
elements for table header ...  
ELEMENT CLOSE "thead.  
ELEMENT OPEN "tbody.  
END SECTION.  
  
SECTION  
DATA  
    USING row1 compname, hzname, etc.  
END SECTION.  
  
SECTION WHEN AT END  
DATA  
    ELEMENT CLOSE "tbody".  
    ELEMENT CLOSE "table".  
    ELEMENT CLOSE "section".  
END SECTION.
```

Each ELEMENT OPEN must have a matching ELEMENT CLOSE.

## Column Specifications

### Syntax:

$$\text{column\_spec} \Rightarrow \left. \begin{array}{l} \text{literal} \\ \text{identifier} \\ \text{FIELD} \\ \text{PAGE} \\ \text{PAGES} \\ \text{array\_spec} \end{array} \right\} [\text{column\_layout}]$$

$\text{array\_spec} \Rightarrow \text{ARRAY} (\text{column\_spec} [ , \text{column\_spec} ] \dots )$

The column specification identifies exactly what will be printed at a particular spot in a report. A column can print data from a literal, variable, data element, or page number. It can also be a compound column (ARRAY) consisting of one or more sub-columns. In a template definition, the keyword **FIELD** is used as a place holder, with the actual element, variable, or literal to be supplied later.

If a variable or element is printed, its value at each report cycle prints according to the layout options. If a literal is used, it prints the same value each time. The keywords **PAGE** and **PAGES** generate page numbering, and are normally used in headers or footers. Wherever the word **PAGE** occurs, the number of the current page is substituted, before column layout options are applied. The keyword **PAGES** is replaced by the total number of pages in the report, as in "Page n of m".

When the **ARRAY** specification is used, a group of one or more columns is printed repetitively, with the same format. Array columns are used only with crosstab reports. The number of columns actually printed equals the number of column values in the crosstab, times the number of column specs in the **ARRAY** spec. The printing sequence is to print all the columns listed in the **ARRAY** spec, then repeat for the number of crosstab values. Any column layout options listed outside the parentheses of an **ARRAY** spec apply to all columns within the parentheses, unless overridden by layout options inside the parentheses which apply to an individual column.

In the description of the **EXEC SQL** Statement, there is an example of a crosstab on page 27. It produced the data shown in the following table. The variables *areaname* and *acres* are arrays with 3 values each.

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500

10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

The following example shows one way these data could be printed. Ignoring column formatting details for the moment, these line specifications for heading and data would produce the report fragment shown.

HEADING

AT 1 musym LABEL, muname LABEL, ARRAY(areaname).

DATA

AT 1 musym, muname, ARRAY(acres, "acres").

musym	muname	X	Y	Z
10A	Alpha loam, 0 to 3	100 acres	300 acres	400 acres
10A	Alpha loam, 0 to 3	200 acres	acres	500 acres
10B	Alpha loam, 3 to 6	600 acres	700 acres	acres
10B	Alpha loam, 3 to 6	acres	800 acres	acres

The heading line prints the labels for the data elements *musym* and *muname*, which we assume are just the column names, then the values for *areaname*, which define the groupings.

The data line prints *musym* and *muname*, this time as normal report columns, then *acres* and the literal "acres" as an array. The values from *acres* are paired with the word "acres" and printed in three columns. In this example, the crosstab was not set up with aggregation, so there are several blank spaces, but the literal prints anyway. The report could be made to look better by changing the crosstab, or moving the word "acres" into the heading.

When a multiple valued variable is printed in a column that does not have an array spec, the values are printed one beneath the other in the column. It results in a set of parallel report columns for each query column, as illustrated earlier.

## Column Layout Specifications

### Syntax:

```
column-layout ⇒ [ WIDTH number [IN] ]
                [ WIDTH UNLIMITED ]
                [ LABEL ]
                [ DIGITS number ]
                [ DECIMAL number ]
                [ SIGDIG number ]
                [ ALIGN { LEFT | CENTER | RIGHT } ]
                [ PAD "character" ]
                [ INDENT number [IN] ]
                [ NEST number [IN] PER identifier ]
                [ NO COMMA ]
                [ TRUNCATE ]
                [ REPEAT ]
                [ SEPARATOR "string" ]
                [ REPLACE NULL [WITH] literal ]
                [ REPLACE ZERO [WITH] literal ]
                [ SUPPRESS [DUPLICATES] [ BY identifier ] ]
                [ QUOTE[D] [ quote-string ] [ ESCAPE escape-string ] ]
                [ TAG "name" [ attribute ] ... ]
                [ VALUETAG "name" [ attribute ] ... ]
```

```
attribute ⇒ ATTRIB ( "string", { "string" | variable } )
```

Each column in a report can use zero or more of the above layout options. Each option can be used only once per column. The options are generally the same for headings and data, although some are not useful in headings. The options can be written in any order:

1. The **WIDTH** option overrides the default width for the data in the column. The default width is taken from the template in a **USING** statement, from the data dictionary for data elements, or from the string length for a literal without the **REPEAT** option. In text style output there is no default width for a variable, but in XML output the default is **WIDTH UNLIMITED**. If the width is followed by **IN** (or **INCH** or **INCHES**), the width is measured in inches as determined by the horizontal pitch. The default is to measure width in characters.
2. The **WIDTH UNLIMITED** option formats the output without fixed column widths. This overrides the normal word wrap function, as well as the **TRUNCATE**, **ALIGN**, **INDENT**, **NEST** and **REPEAT** formatting options. The data for the column is printed in the minimum space needed to contain the entire value, preceded by the optional **SEPARATOR** string. Numbers are formatted with decimal places defined in the usual manner, and with no leading spaces or zeros. This is useful with the

PAGE WIDTH UNLIMITED option for producing text style exports, and is the default for XML style output.

3. When LABEL is specified, the value printed is not the data, but the *ColumnLabel* from the data dictionary for the specified element. This could be used in column headings. If LABEL is used with a literal or variable, the result is a blank.
  
4. The DIGITS option is used with numeric data to specify the number of digits to be printed to the left of the decimal point. The default number of digits for data elements is taken from the data dictionary. This specification is overridden if the WIDTH is given explicitly. Numeric values over 999 are printed by default with commas between groups of 3 digits. The commas are not counted as digits, but do count in the column width.
  
5. The DECIMAL option is used with numeric data to specify the number of digits to be printed to the right of the decimal point. The default number of decimal places is taken from the data dictionary if a data element is being printed, otherwise the default is zero. If the number of decimal digits is zero, the decimal point is not printed.
  
6. The SIGDIG option is used with numeric data to specify the number of significant digits in the value to be printed. The value is rounded off so that only the significant digits are shown, and zeros are added as necessary to fill out the remaining places required by the DIGITS and DECIMAL specifications. The number of significant digits specified must be greater than zero, and if SIGDIG is not specified, all digits are considered significant. The following examples show the relationship of the DECIMAL and SIGDIG specifications:

Original Value	DECIMAL	SIGDIG	Result
527.36	2	3	527.00
0.456	2	1	.50
1384.2	0	2	1400

7. The ALIGN option positions the data within the column. The default is based on the data dictionary definition for elements. For variables and literals the defaults are left alignment for character data and right for numeric.
  
8. The PAD option provides a character to fill out blank space in the column when the data is shorter than the column width. Padding occurs on the right if the column is aligned left, on the left if the column is aligned right, and on both sides if the column

---

is aligned center. If the text in a column is word wrapped, padding is only applied on the last line of the text. The default pad character is a space.

9. The **INDENT** option positions the data a specified number of characters or inches from its alignment position. A positive indent applies to the first line of a word wrapped string, while a negative indent applies to lines after the first. In other words, for typical left aligned data, a positive indent produces first line indentation, while a negative indent produces “hanging” indentation. For right aligned data, it works the same way but relative to the right edge of the column.
10. The **NEST** option is provided for printing interpretations. These are traditionally printed in a “nested” or “outline” format, with results of each sub-rule indented below its parent rule. The amount of indentation increases with each level of sub-rule. So a nested format is really a variable indentation, with the amount of indent proportional to the depth of nesting. The output of the **INTERPRET** command includes a column *InterpRuleDepth* for just this purpose. The **NEST** format option allows you specify an indentation of *n* spaces (or inches) per depth level. This can be combined with normal indentation, such as a negative indent amount for hanging indents. For example, to print an interp result *RatingClassNameHighRV* with hanging indent of 1 space for word wrapping, plus nesting of 2 spaces per level, use:

```
RatingClassNameHighRV INDENT -1 NEST 2 PER InterpRuleDepth
```

11. The **NO COMMA** option suppresses the placement of commas in numbers larger than three digits. This is used when printing a numeric value that should not have commas, such as a year or an ID number. It is also used when exporting data in a comma delimited format, to avoid inappropriate commas.
12. The **TRUNCATE** option determines what happens when character type data is too long to fit in a column. The default is to split the data across multiple lines with word wrapping. If **TRUNCATE** is specified, the data is printed on a single line and truncated to the fit the column width. Numeric type data is never wrapped; if it is too long to fit the column, asterisks are printed.
13. The **REPEAT** option means that the column’s value is repeated as often as necessary to fill the column width. This is typically used with a literal, such as “-”. The **REPEAT** option in this case would fill the column with dashes. The column width must be specified explicitly when **REPEAT** is used.
14. The **SEPARATOR** string prints before the data for the column prints. It is placed at the column’s specified starting position, and the actual data for the column starts after the separator. If this option is not included, no separator is printed. There is no way to specify a separator to the right of a field. To print a right border on a line, add a

field of width zero at the end of the line, with the desired border character as its separator.

15. The **REPLACE** options allow the printing of some other value when a zero or null is found. This does not affect the operation of any calculations based on the value being replaced. This function can also be achieved using a variable with a conditional expression, but the **REPLACE** form might be more convenient. A value set to null by **SUPPRESS DUPLICATES** is not replaced with the substitution value, but always prints blank.
16. The **SUPPRESS DUPLICATES** option prevents repetitive printing of data. For each report input record, the value of a column specified with **SUPPRESS** is compared to its value in the previous record. If it matches, blanks print instead of the value. If the column is part of the sort key for the main report query, the duplicate suppression does not occur on control breaks. In this context, a control break occurs when the column or any column higher in the sort key changes value.

This control break behavior can be obtained for non-sort columns with the **BY** phrase. The identifier after **BY** is an element or variable to be tested if the value of the column itself does not change. If there is a change in the value of the **BY** variable (or higher sort columns if the **BY** variable is in the sort key), suppression does not occur.

17. The **QUOTE** or **QUOTED** option surrounds the column's data with quotation marks and escapes any embedded quotation marks. This is typically used when exporting data to another program. The **quote-string** is a single character that will be added to the beginning and end of the data. It defaults to the quotation mark ("). The **escape-string** is another single character whose default is the back-slash (\). If the data contains an occurrence of the **quote-string** or the **escape-string**, it will be preceded in the output by the **escape-string**. If the **quote-string** and the **escape-string** are the same, it means that embedded quotes will be doubled (which is the SQL convention). To specify a quotation mark, surround it by single quotes: "'".

For example, to use the single quote instead of the double quote, and to double the quote if it appears in the text, the format option would be written as:

```
QUOTE ` ` " ESCAPE ` ` "
```

If the original text contained the following:

```
Elmer said, "That's all, folks."
```

The output using the above format would be:

```
'Elmer said, "That''s all, folks."''
```

18. The **TAG** option is available in XML style output only. It surrounds the column's data with XML opening and closing tags using the "name" string as the tag name. Additional attributes can be specified with the **ATTRIB** keyword followed by a name and value in parentheses. The attribute name must be a quoted string and the attribute value may be a quoted string, a number or a defined variable. The tag name and attributes are output according to XML standards.

If the variable being printed in the column is an array with more than one value, the tag surrounds the full set of values. The result is that the values are concatenated together unless a **VALUETAG** option is used to apply a tag to each individual value.

The **TAG** option must be used within an **ELEMENT**. Examples are shown in [ELEMENT](#) description.

19. The **VALUETAG** option is available in XML style output only. It is similar to the **TAG** option except that it places tags around the individual values in the array of values for the column. If the column's variable has just one value, the **TAG** and **VALUETAG** have the same effect. If both are used on a column, the **VALUETAG** appears inside the **TAG**.

## SET

### Syntax:

```
SET column_name [ FROM variable ] [, column_name [ FROM variable ] ] ... .
```

### Used In:

#### Calculation

### Example:

```
SET aashind_l, aashind_r, aashind_h.  
SET dbfifteenbar_r FROM db.
```

The **SET** statement is used in calculation scripts to store the results of a calculation back to the database. The value of the FROM variable is placed in the specified column. If the column and FROM variable have the same name, the “FROM variable” part may be omitted. You may use multiple **SET** statements or multiple columns in a single **SET** when the calculation script produces more than one result. The results are stored for each row the user chooses to be calculated. Rows can be modified only if they are editable and checked out by the user. If the specified column contains manually entered data (flagged as “M”), or data from prior to the existence of a calculation (flagged as “P”), it is not changed unless the user chooses to override (in the Calculation Manager dialog).

Values can be stored in two ways: singly or in groups. If the *column\_name* is a column in the base table of the calculation, a single value will be stored in each calculated row. If the source variable has more than one value, only the first one is used. If the calculated variable has a null value, a null is stored in the column.

A group of values can be stored by specifying a column in a table that is a direct child of the base table. This causes all existing child rows in the selected set to be updated. If necessary, rows will be added or deleted to match the number of values in the source variable. More than one column in the child table may be given new values, by using multiple **SET** statements or multiple columns in one **SET**. Case should be taken to see that all source variables have the same dimension, or data could be lost.

In case of ambiguity in column names, the form *table.column* may be used for *column\_name*. The *\_r*, *\_l*, or *\_h* suffix must be used if the column has modal values.

## TEMPLATE

### Syntax:

TEMPLATE template-name [ column-layout ] output-specification .

template-name  $\Rightarrow$  *name*

### Used In:

#### Report

### Example:

```
TEMPLATE basic SEPARATOR "|"
AT LEFT FIELD WIDTH 8, FIELD WIDTH 50.
```

A template describes the format of a report line without the data. Templates are not required, but are useful to avoid repetitive specification of layout options. Putting the statement "USING template-name" into an output specification copies all the column layout information from the template into the output specification.

In a template, a set of column layout options can be given right after the template name, and these will be the default for all columns in the template. There must be one and only one output specification in a template, which must be either an **AT** or an **ELEMENT** statement. This can contain additional column layout options, which take precedence over the template defaults. Finally, when a template is invoked with a **USING** statement, other layout options can be given, which take precedence over the template. Column and line specifications are described under the **SECTION** statement.

In the output specification used in a template, it is possible to use a literal, variable or element name as a value to be printed in some column. This would print the specified value whenever the template is used. However, the keyword **FIELD** can also be used in place of a value, which means that the value to be printed is not defined until it is specified in a **USING** statement. An output specification in a template definition may not contain **USING**.

**WHEN****Syntax:**

WHEN expression DISPLAY message [ parameter [, parameter ] ... ].

**Used In:****Validation****Example:**

```
WHEN sum_pct > 100 DISPLAY "Percents sum to more than 100".  
WHEN error DISPLAY "Error in horizon %s" hzname.
```

The **WHEN** statement is used in validation scripts to produce a message when an error condition is detected. The expression after **WHEN** is evaluated for each row to be validated, and if a True (non-zero) value is found the message is added to the validation message list. If the message contains substitution markers as used in *sprintf* (such as %s or %g) values are taken from the list of parameters and placed into the message. The validation process also records information about which row generated a message, and this is included when the message list is displayed.

In some cases it is useful to have multiple values for the **WHEN** expression, or for the message or its parameters. This causes multiple messages to be generated for each row validated. If the validation script extracts data from a child of the base table, individual messages for each child row can be produced by using parameters that have values collected from the child rows.

### Using Subreports

The purpose of a subreport is to produce some output that is loosely coupled to the primary report, meaning that a subreport has its own set of queries and output specifications that might not be related to those of the primary report. It allows for greater flexibility in cases where complex formatting is required.

A subreport is requested with an `INCLUDE` statement in the data block of an output section. The entire output of a subreport is inserted in the data block as a single logical line. If keep processing is in effect, it will attempt to keep the subreport output together on a single page. Therefore it is advisable to design subreports so their output is less than a page. Longer output will spill over onto additional pages of the main report and possibly produce unwanted results. However, it is also possible to have a main report that produces no output of its own and only calls a series of subreports, in which case the main report will be a page by page copy of the subreports.

Subreports may not specify any page layout, such as the page size, font, headers or footers. The page layout of the main report controls all output from subreports.

A report and its subreports do not need to use the same base table, and no automatic synchronization is done as with properties in a `DERIVE` statement. Subreports may call themselves in a recursive fashion to produce a report on recursively organized data. An example is a report to list rules and all their subrules at any depth. It is important to pass the right parameters to a subreport so that it will find the right records to report on and not get into an endless recursion.

## Appendix 1: Conventions for HTML Reports

### DocBook XML

This appendix describes the conventions for reports used to create HTML reports like those used in the Web Soil Survey. If you want to produce a report in that style, the report script must produce XML output that conforms to the DocBook XML standard. A DocBook reference manual is available at <http://www.docbook.org/tdg/en/html/docbook.html>.

Within the broader world of DocBook applications, we are using just those features needed for Web Soil Survey reports. Since WSS reports are intended for use within a larger soil survey document, the outermost element of a report is the `<section>`<sup>1</sup>. That `<section>` uses the attribute `label="SoilReport"` to identify it. Furthermore, the DocBook standard requires that every

---

<sup>1</sup> In the narrative we use the XML convention of angle brackets to designate an element, such as `<section>`. In examples of report scripts the element names are in quotes, using the report language syntax.

---

<section> must include at least a <title> element. All of these requirements can be met by including a standard **AT START** section in a report script, such as:

```
SECTION WHEN AT START
DATA
  ELEMENT OPEN "section" ATTRIB ("label", "SoilReport").
  ELEMENT "title" reporttitle.
END SECTION.
```

In the above example, notice the syntax of the **ELEMENT** command. If the word **ELEMENT** is followed by **OPEN** or **CLOSE**, the output will contain only the opening or closing tag of the element. If not, a complete XML element with opening and closing tags is produced. If **ELEMENT OPEN** is used, there must be a corresponding **ELEMENT CLOSE** somewhere in the report script (with exceptions as noted below).

The first **ELEMENT** line is the beginning of the outermost section, as described above. The report language syntax requires that there be quotes around the tag name (“section”) as well as the attribute name (“label”) and value (“SoilReport”). An element can have more than one attribute, by simply adding more **ATTRIB ( )** specifications.

Because this line only opens the <section>, there must be a closing tag as well. Normally this is handled using a **SECTION WHEN AT END** block in the report script, such as:

```
SECTION WHEN AT END
DATA
  ELEMENT CLOSE "section".
END SECTION.
```

The line beginning **ELEMENT “title”** is a complete element definition and produces the required <title> for the <section>. In WSS reports the title of the outermost section is the name of the report or table. This example prints the contents of a script variable named `reporttitle`, which will be explained later.

## XML Elements Used in Reports

After the **AT START** section the report script will contain a number of **SECTION** blocks to specify the contents of the report. The **ELEMENT** command is the main output formatting control and is used either in a **TEMPLATE** or directly in a **SECTION**.

The **ELEMENT** command can contain a list of data fields, each of which can have a **TAG**, **VALUETAG**, and attributes. Tags and valuetags generate complete XML elements that are placed between the beginning and ending tags of the main element. This creates a structure where the outer element can contain data or other elements. The difference between a **TAG** and a **VALUETAG** appears when displaying a report variable that has multiple values. A **TAG** surrounds the entire set of values for the variable, while the **VALUETAG** surrounds each individual value of the variable.

Some examples will illustrate the way **ELEMENT** can be used.

```
ELEMENT "title" "Mapunit Description".
```

---

This is the simplest case, consisting of an element name and a string of data. It translates to the following XML:

```
<title>Mapunit Description</title>
```

ELEMENT "col" ATTRIB ("width", "3\*").

An element can have attributes as well as data. Attributes become part of the opening tag for the element. In this case, the element has no data so the opening and closing tags get condensed into a single tag:

```
<col width="3*" />
```

ELEMENT "tr" ATTRIB ("class", "heading")  
FIELD TAG "td" ATTRIB ("class", "begindatagroup"),  
FIELD TAG "td" ATTRIB ("class", "enddatagroup").

This example shows the use of TAG to create elements within an element. Each of these can have attributes of their own. The first ATTRIB goes with the <tr> element, and the others go with their adjacent <td> elements.

Notice the syntax for this example. The content of an element follows the element name and optional ATTRIB. In this case the content is two FIELDS, so this example would have to be inside a TEMPLATE. The keyword TAG follows FIELD to indicate that the field will be surrounded by tags to create a <td> element. The comma separates the two fields in the <tr> element.

If this example appeared in a TEMPLATE, there would have to be a USING statement to specify the data to be printed. This looks like a heading, so the data might be "Plant Name" and "Pct. Composition". Then the XML would be:

```
<tr class="heading"><td class="begindatagroup">Plant Name</td>  
<td class="enddatagroup">Pct. Composition</td></tr>
```

ELEMENT "tr" ATTRIB ("class", shading)  
hzname TAG "td" VALUETAG "para",  
hzdept\_r TAG "td" VALUETAG "para" ATTRIB ("role", "number").

This example prints data from variables that have multiple values. We can assume that the query in this report used AGGREGATE by component and specified NONE as the aggregation type for the horizon names and depths (hzname and hzdept\_r). The TAG "td" places each of these variables in a <td> element, which represents a table cell. Then each individual value is placed in a <para>, which stands for paragraph and means that each will appear on a separate line of output. Thus the horizon names and depths will be listed as side by side columns, although each column is actually within a table cell. This is a common style for soils reports.

Notice the attributes in this example. The <tr> has an attribute value that is not in quotes. It refers to a variable named shading, which would have been created in a DEFINE statement to hold some appropriate class attribute, such as "even" or "odd" (see table of attributes

---

below). Also, the hzname has no attribute, so it gets the default format, which is simply text left justified in the cell.

Assuming some arbitrary data for shading, hzname and hzdept\_r, this would create the following XML. This has been indented to make it more readable, but the indentation is not part of the actual report output and has no meaning:

```
<tr class="odd">
  <td>
    <para>A</para>
    <para>B</para>
    <para>C</para>
  </td>
  <td>
    <para class="number">0</para>
    <para class="number">15</para>
    <para class="number">75</para>
  </td>
</tr>
```

```
ELEMENT "variablelist" VALUETAG "varlistentry"
  hzname TAG "term",
  hzdept_r TAG "listitem" VALUETAG "para".
```

This example shows the same data as the previous example in a semi-tabular format rather than in a table. It uses the DocBook element `<variablelist>` which can be used to create bullets or numbered lists, depending on the final style choices. In the report script we do not need to be concerned with the formatting details, only the structure.

A `<variablelist>` contains a number of `<varlistentry>` elements, each of which contains a `<term>` and a `<listitem>`. The `<listitem>` must contain some other element, in this case a `<para>`. To accomplish this there has to be a **VALUETAG** at the element level. This means that the `<varlistentry>` element is produced for each set of values within the element. An additional **VALUETAG** on the `hzdept_r` produces a `<para>` around the horizon depth. The resulting XML is thus four levels deep:

```
<variablelist>
  <varlistentry>
    <term>A</term>
    <listitem>
      <para>0</para>
    </listitem>
  </varlistentry>

  <varlistentry>
    <term>B</term>
    <listitem>
      <para>15</para>
    </listitem>
  </varlistentry>

  <varlistentry>
    <term>C</term>
    <listitem>
      <para>75</para>
    </listitem>
  </varlistentry>
</variablelist>
```

---

```
</listitem>
</varlistentry>
</variablelist>
```

## Elements used in tables

The DocBook standard includes the HTML standard for creating tables, which will be familiar to most people who have created web pages. The basic elements of a table are listed here. All of them are needed for a complete WSS report, but for a report that is not for publication many of the elements can be left out.

- `<table>` encloses the whole table.
- `<title>` is for a title that is printed above the table. In WSS reports this is the name of the soil survey area. If the report uses national mapunit symbols there is no survey area and the table title is omitted.
- `<col>` specifies attributes for a column in the table. If used, there must be one `<col>` element for each column. In WSS reports this is used to specify a relative width for the column.
- `<thead>` is for the table header. Rows within a `<thead>` receive special formatting so they look like a heading. They are also repeated at the top of each report page when the report is converted to PDF.
- `<tbody>` is the body of the table and occurs after the `<thead>`.
- `<tr>` is a row of a table. It can be used within a `<thead>` or a `<tbody>` and it contains one or more `<td>` elements for the data in the row.
- `<td>` is table data, sometimes called a cell of a table. It must be within a `<tr>` element.
- `<para>` is a paragraph that can be included in a `<td>`. It is not required, since data can be included within a `<td>` directly. The `<para>` is often used in WSS reports to identify the type of data so that standard formatting styles can be applied.

## Elements used in non-table reports

Reports that are not formatted as a table include the various Map Unit Description reports, the Interpretation Description, and similar reports. Although there is no `<table>` element there is an internal structure to these reports, and the `<section>` element is generally used to represent it. The DocBook standard allows sections to be included within sections. So, for example, if a report is organized by survey area there will be a `<section>` for each new area name. A Map Unit Description report will have a `<section>` for each mapunit within the area, and a variety of `<section>`s within the mapunit. Each `<section>` must have a `<title>`, which is used as a heading for the section when it is displayed.

---

Data in these reports is typically displayed in a list (semi-tabular) format. The DocBook <variablelist> element is used to create a list structure, which can then be displayed in various ways. The final example above provides an explanation of the use of <variablelist>.

Reports that contain large blocks of text will use the <para> element to enclose the text. The way you expect the text to be stored in the database will influence the attributes applied to the <para>. If the text is simply a single paragraph no special attributes are needed. If the text contains line breaks, tabs or spaces that need to appear in the output, ATTRIB (“role”, ”preservenewlines”) is needed. If the text contains XML markup that needs to be preserved in the output, use the SPECIAL option, as in:

```
textdata TAG "para" SPECIAL.
```

### Attributes used in all elements

Attributes describe the type of data or the role played by an element in the overall report structure. This information is used when formatting the report for display, and it can be used in different ways depending on the display format. For example, the same report might appear in a web site and in a PDF document meant for printing. This separation of the functional description of a report and its display formatting allows one report script to serve multiple purposes. However, only certain attributes are recognized by the formatter, and unrecognized attributes are ignored. So the conventions listed here must be followed closely to produce reports with a consistent appearance.

Element	Attribute Name	Attribute Value	Meaning
"section"	"label"	"SoilReport"	Required to identify the outermost section of a soil report
		"Survey_Area"	Data for one survey area. The title of the section is the survey area name. Used in reports that don't have tables.
		"Map_Unit_Description"	Data for one mapunit in a map unit description report.
		others	The Map Unit Description report has labels to signify the type of data in each section, but they are currently ignored by the formatting program,
"title"	"role"	"suppressTitle"	Do not display a title for the current section. Although the <title> element is required in a <section>, this will suppress display of the title.
"table"	"orient"	"land"	Table is wide and should be displayed in landscape orientation.
"col"	"width"	"n*"	Relative column width expressed as a number followed by an asterisk. A column with width "3*"

			is 3 times as wide as a column with width "1*". The exact width of a column depends on the overall width of the table and width of the other columns.
"tr"	"class"	"mapunit"	A table row that begins the data for a map unit.
		"even"	A table row that can be shaded in alternating colors to improve readability. Alternates with "odd".
		"odd"	A table row that can be shaded in alternating colors to improve readability. Alternates with "even".
		"interpdata"	A table row containing data about an interpretation in the Survey Area Data Summary report.
		"units"	A table row containing units of measure. It is a type of subheading for the table
"td"	"rowspan"	a number	Number of rows in the heading occupied by this cell. Part of standard HTML tables.
	"colspan"	a number	Number of columns in the heading occupied by this cell. Part of standard HTML tables.
	"class"	"begindatagroup"	Cell is the first of a group of cells that are set off visually by a heavier vertical border on the left edge.
		"enddatagroup"	Cell is the last of a group of cells that are set off visually by a heavier vertical border on the right edge.
		"datetime"	Cell contains a date/time field that should be formatted according to the date conventions for the report.
"para"	"role"	"mu-name"	Content is a mapunit symbol or name.
		"comp-name"	Content is a component name.
		"number"	Numeric data, normally displayed right justified in a cell.
		"class-name"	Non-numeric data, such as a class name, normally displayed centered in a cell.
		"hang-list"	Multiple values of character type data which are displayed in a vertical list with hanging indents

		"preservenewlines"	Content is a text field that may include newlines, tabs, and significant spaces. Normally this "white space" is removed. This attribute will preserve the layout of the original text.
--	--	--------------------	--

## Parameters for Web Soil Survey Reports

A general report script can use any parameters, but reports for WSS must use the parameters which that system supplies. In general, a user in WSS can choose either a full soil survey area or a set of map units to report on. These choices will be passed to the report script. Other parameters are used in specific reports. The parameter names and types must be defined as in the list below. The report script can also define a prompt for each parameter, which will appear in the user interface for entering parameter values when applicable.

### **PARAMETER areasym CHAR.**

The area symbol for the requested survey area. If selecting by map unit, this will be null.

### **PARAMETER mukeys MULTIPLE CHAR.**

The map unit keys for the selected map units. If selecting by survey area this list will be empty.

### **PARAMETER includeminor BOOLEAN.**

Set to 1 if minor soils are to be included in the report, or 0 if they are to be excluded.

### **PARAMETER useNationalMapunits BOOLEAN.**

Set to 1 if national mapunit symbols (mukey) are to be printed, or 0 if survey based mapunit symbols (musym) are to be used.

### **PARAMETER crops MULTIPLE CHAR.**

The list of crop names selected for inclusion in a crop yield report.

The last example uses what is called a "dynamic choice list", meaning that the list of choices depends on the area of interest selected by the user. There is currently a limited set of parameters that can be used as dynamic choice lists. In place of "crops" the following parameter names can be used:

**cocrops:** Crop names from the Component Crop Yield table that have non-null yield data.

**mucrops:** Crop names from the Mapunit Crop Yield table that have non-null yield data.

**interps:** Names of interpretations (rules) exported with the selected survey area(s).

**sainterps:** Same as **interps** but allows more than 3 selections.

**cotextkinds:** Component text kind and category.

---

**mutextkinds:** Mapunit text kind and category.

**nontechdesccats:** Mapunit text categories where text kind is “nontechnical description”.

**counties:** County names found in the Legend Area Overlap table for the area of interest.

## Using parameters in a report query

Most of the parameters listed above are used in the report queries to control the selection of data. In one case, “useNationalMapunits”, the parameter is also used in the body of the report to alter parts of the output. In the queries, some unusual SQL features are used. This is an example from a crop yield report that uses most of the parameter options:

```
1) EXEC SQL select areaname, musym, museq, mapunit.mukey muiid, muname,
2) Nationalmusym, compname, comppct_r, localphase, component.cokey,
3) cropname, yldunits, irryield_r, nonirryield_r
4) from legend, mapunit, component, outer cocropyld
5) WHERE join legend to mapunit
6) and join mapunit to component
7) and join component to cocropyld
8) and ( areasym=$areasym OR mapunit.mukey in ($mukeys) )
9) and ( majcompflag = "Yes" OR 1 = $includeminor )
10)and cropname in ($crops)
11)ORDER BY case when $useNationalMapunits =0 then areaname else
    nationalmusym end,
12)museq,
13)cropname;
14)AGGREGATE ROWS muiid
15)COLUMN yldunits UNIQUE GLOBAL
16)CROSSTAB cropname VALUES (crops)
17)CELLS irryield_r, nonirryield_r, yldunits.
```

Line 8 uses the “areasym” and “mukeys” parameters to select the map units for the reports. By using OR, it will select either by survey area or by mapunit, depending on which parameter is supplied. Note that when testing this report, if you enter values for both parameters it will select mapunits that meet either criterion, which may be more than you expected.

Line 9 uses the “includeminor” parameter to select minor components. Here OR is used to say, select a component if it is a major component, OR if the includeminor parameter is set to 1.

Line 10 uses the “crops” parameter to select crop names. The parameter is used again in line 16 to specify the crosstab values. Doing this forces the report to include a column for each crop, even if there is no yield data for a crop. It also forces the columns to appear in the order that the crop names are entered in the parameter.

---

Line 11 makes use of the “useNationalMapunits” parameter to control the sorting of the report. This is tricky because when survey area mapunit symbols are used the report is sorted by survey area, but when national mapunits are used it is not. The SQL CASE expression is used here, which is similar to the IF expression in a DEFINE statement. Line 11 says to sort by areaname or by nationalmusym depending on whether survey area or national symbols are used.

Line 12 says to sort on museq. The column museq is the sequence number for mapunits within a legend, so it is used to sort when using survey area map units. It has no effect when using national mapunit symbols, but it does no harm to have it there.

## Script Variables

A script variable is a special notation used to obtain information about the report script itself from the database. An example is the name of the report script, which is stored in the “report” table in the column “report\_name”. To include this name in the output of a report, use a reference to the script variable SCRIPT\_NAME, as in:

```
DEFINE reportname INITIAL SCRIPT(SCRIPT_NAME).
```

This places the script variable into a normal report variable, reportname, which can then be printed as part of the report heading. Using INITIAL in this statement means that the variable reportname is set at the beginning of report processing and never changes.

The script variables available in NASIS are a little different from the ones in WSS (from the Soil Data Mart database) due to differences in table structure. The script variables are:

<b>SCRIPT_NAME</b>	The name of the report.
<b>REPORT_TITLE</b>	NASIS: same as SCRIPT_NAME. WSS: the report title from the Home tab in the report editor.
<b>REPORT_HEADER</b>	NASIS: blank. WSS: The text to be used as a head note in the report, from the Home tab in the report editor.
<b>INTERP_NAME</b>	NASIS: blank WSS: A list of rule names to be included in an interpretation report, from the Interpretations tab in the report editor.
<b>INTERP_TITLE</b>	NASIS: blank. WSS: The list of column headings to be used with the corresponding rule in the INTERP_NAME list. This is also entered in the Interpretations tab.
<b>SCRIPT_ID</b>	The internal ID number for the record in the report table. This might be used to query for data linked to the report in the database.

---

Using script variables allows a report script to be generalized, so that certain features do not have to be coded into the script. This capability is used for WSS interpretation reports, which use a completely generic script. All standard interpretation reports in WSS use exactly the same script. That script uses script variables to pick up the report name, head note, and list of interpretations from data stored in the report table. To create a state report using custom interpretations you can simply copy an existing interpretation report and change the list of interpretations and titles in the Interpretations tab. You also specify which states will use the report in the Usage tab.

---

## **NASIS CVIR Script Writing References**

### **Database Structure Guide**

The NASIS 6.0 Database Structure Guide is a comprehensive reference that describes all aspects of the NASIS database design. The guide provides information you need to know about the NASIS template model, naming conventions and data types. It can be obtained from the NASIS web site: <http://nasis.usda.gov/documents/metadata/6.0>.

### **Table Structure Report**

The Table Structure Report is included in the *NASIS 6.0 Database Structure Guide*. The table structure report provides information you need to know about table and column physical names, modality, data types, and other characteristics necessary for report writing.

### **Database Structure Diagrams**

The Database Structure Diagrams are included in the *NASIS 6.0 Database Structure Guide* and the NASIS Online Help. Because of the size of the database the diagrams each show just one object hierarchy. They show the table relationships required for completing joins between tables.

## Index

ABS .....	31	CODEVAL .....	59
ACCEPT .....	<b>10</b> , 11, 34, 39	COLUMN (aggregation).....	42
ACOS .....	<i>See</i>	compute a sum .....	27
AGGREGATE ROWS BY .....	42	compute an average .....	28
aggregation .....	32, 42, 45	concatenation .....	18, 20
aggregation functions.....	42	conditional expression .....	14
alias.....	37	control break.....	80
ALIGN.....	77, 79	controlling column.....	45
ALL .....	14	conventions .....	9
ANY .....	14	convert codes .....	22, 23
APPEND.....	23	convert geomorphic descriptions .....	19
arithmetic expression .....	14	convert texture codes .....	19
array.....	13	convert to name case.....	19
array dimension .....	44, 45	convert to sentence case.....	19
ARRAYAVG.....	28	convert to upper case .....	18
ARRAYCAT .....	20	COS .....	30
ARRAYCOUNT.....	23	COUNT .....	26
ARRAYMAX .....	24	covert to lower case .....	18
ARRAYMIN.....	23	cross products .....	40
ARRAYPOSITION .....	25	<i>crosstab</i> .....	45, 75
ARRAYROT .....	25	CROSSTAB.....	42
arrays .....	43	DATA .....	62
ARRAYSHIFT .....	24	data type.....	96
ARRAYSTDEV .....	28	Database Diagram.....	96
ARRAYSUM.....	27	Database Structure Guide .....	96
ascending .....	41	DATEFORMAT .....	21
ASIN.....	30	DECIMAL .....	77, 78
ASSIGN.....	<b>12</b>	default aggregation .....	43
AT.....	67	default sort type .....	41
ATAN .....	30	DEFINE.....	<b>12</b>
ATAN2.....	31	delimiter.....	50
average.....	28	de-normalized .....	40
AVERAGE .....	29, 32, 42, 44	DERIVE .....	11, <b>34</b>
base table .....	10, 34	descending .....	41
BASE TABLE .....	<b>11</b>	dialog .....	57
BOOLEAN.....	58	different hierarchic paths .....	40
boolean expression .....	14	DIGITS.....	77, 78
BOTTOM .....	54	dimension.....	13
braces.....	9	direction of sorting.....	41
brackets.....	9	division by zero .....	17
calculation.....	9	EDIT .....	38
calculation scripts .....	82	ELEMENT .....	71
call a property script .....	34	ELEMENT (parameter) .....	57
calling script .....	34	ellipsis.....	9
CELLS (crosstab) .....	42	END SECTION .....	62
CHARACTER .....	58	escape-string .....	80
character spacing .....	61	Evaluations .....	12
character strings.....	39	EXEC SQL .....	<b>36</b>
CLIP .....	18	EXP .....	29
CODELABEL.....	23	export.....	78
CODENAME.....	22, 59	expressions.....	37
CODESEQ .....	59	FILL.....	67

FINAL.....	49	maximum value.....	24
FIRST.....	32, 42, 44	MIN.....	26, 32, 42, 44
FIRST n.....	41	minimum value.....	23
floating point.....	39	MOD.....	31
FONT.....	<b>48</b>	modal.....	39
FOOTER.....	<b>49</b>	modality.....	96
format a report line.....	83	MULTIPLE (parameter).....	58
FROM clause.....	37	multiple input values.....	43
fuzzy values.....	53	multiple valued variable.....	20, 27, 28
generate interpretations.....	51	multiple values.....	13
GEOMORDESC.....	19	NAMECAP.....	22
GLOBAL.....	42, 45	NASIS web site.....	96
global aggregation.....	45	NEST.....	77, 79
HEADER.....	<b>49</b>	NEW.....	22
HEADING.....	62	NEW PAGE.....	49, 67
heading lines.....	64	NMCASE.....	19
HORIZONTAL.....	61	NO COMMA.....	77, 79
IF THEN ELSE.....	14	NONE.....	42
implementation name.....	39	NOT.....	14
INCLUDE.....	67	null value.....	17
INDENT.....	77, 79	OBJECT.....	59
INITIAL.....	14, 49	order of execution.....	34
initial value.....	12	outer joins.....	37
initialization.....	14	PAD.....	77, 79
INPUT.....	<b>50</b>	PAGE.....	<b>55</b>
input file delimiter.....	50	page n of m pages.....	75
insensitive (sort type).....	41	page numbers.....	49
INTERPRET.....	51	PAGE PAD.....	<b>55</b>
INTERVALS.....	47	PARAMETER.....	<b>57</b>
invoke a property.....	34	parametric query.....	10, 39
ISNULL.....	14	physical name.....	96
iteration.....	11, 12, 62	PITCH.....	54, 55, <b>61</b>
JOIN.....	38	POW.....	31
keywords.....	9	printing crosstab arrays.....	76
LABEL.....	77, 78	PROMPT (parameter).....	58
LABELS (crosstab).....	42	property.....	9
LAST.....	32, 42, 44	Property script.....	10
layout option precedence.....	83	purpose of EXEC SQL.....	37
LEFT.....	54	query.....	37
lexical.....	41	QUOTED.....	77, 80
line, logical vs. physical.....	67	REAL.....	38
lines per inch.....	61	recursive reports.....	85
LIST.....	32, 42, 44	REGROUP.....	32
literal.....	14	REPEAT.....	77, 80
local variable.....	34	repeating groups.....	40
LOCASE.....	18	REPLACE.....	21, 80
log files.....	39	REPLACE NULL WITH.....	77
LOG10.....	29	REPLACE ZERO WITH.....	77
logical line.....	67, 85	report.....	9
logical name.....	39, 96	report section.....	62
login name.....	27	reserved words.....	12
LOGN.....	29	RIGHT.....	54
LOOKUP.....	25	rotate values in array.....	25
MARGIN.....	<b>54</b>	ROUND.....	31
MAX.....	26, 32, 42, 44	SEARCH.....	58

SECASE .....	19	template .....	68
secondary aggregation .....	32	TEMPLATE .....	<b>83</b>
SECTION .....	<b>62</b>	temporary tables.....	39
SELECT clause.....	37	test condition.....	14
SELECTED .....	58	TEXTURENAME .....	19
semicolon.....	39, 69	TODAY .....	27
SEPARATOR.....	77, 80	TOP .....	54
SET .....	<b>82</b>	trailing blanks .....	18
shift values in array.....	24	TRUNCATE.....	77, 79
SIGDIG.....	78	type of script.....	9
SIN.....	30	UNIQUE.....	42
SKIP .....	67	UNLIMITED.....	55, 77
SORT BY .....	41	UPCASE.....	18
SPRINTF .....	27	USER .....	27
SQRT.....	31	USING.....	67
string expression .....	15	USING (template).....	83
string formats .....	27	validation .....	9
STRUCTPARTS .....	20	validation scripts.....	84
subqueries .....	39	VALUETAG .....	71, 81
subreport.....	68	variable dimensions .....	10
substring.....	18	variable names .....	10
subtotals.....	29	variable types.....	10, 12
suffix.....	39	variable width field.....	78
sum .....	27	variables of different dimensions.....	25, 28, 32
SUM .....	29, 32, 42, 44	VERTICAL .....	61
SUPPRESS DUPLICATES.....	77, 80	weighted average .....	28
symbol (sort type).....	41	WHEN .....	<b>84</b>
Table Structure Report.....	96	WHERE clause.....	38
TAG.....	81	WIDTH.....	55, 77
TAN.....	30	WTAVG .....	28