THE POWER TO KNOW.

Expression Language 2.7: Reference Guide

SAS® Documentation
August 4, 2017

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *Expression Language 2.7: Reference Guide*. Cary, NC: SAS Institute Inc.

Expression Language 2.7: Reference Guide

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

August 2017

2.7-P1:engelref

Contents

What’s New in Expression Language 2.7: Reference Guide

Overview

The Expression Engine Language (EEL) now includes two new functions to the Information and Conversion category, GEODISTANCE_COSINE and GEODISTANCE_HAVERSINE.

New Information and Conversion Functions

GEODISTANCE_COSINE

Computes the distance between two geographical points. This function provides results faster and provides an acceptable level of accuracy for most cases.

GEODISTANCE_HAVERSINE

Computes the distance between two geographical points. This formula provides greater accuracy, particularly for shorter distances.

Part 1

Expression Engine Language

Chapter 1 Introduction

Overview of the Expression Engine Language

About the Expression Engine Language

DataFlux Data Management Platform is a powerful suite of data cleansing and data integration software applications. You can use the Data Job Expression node to run a scripting language to process your data sets in ways that are not built into the DataFlux Data Management Studio. The Expression Engine Language (EEL) provides many statements, functions, and variables for manipulating data in DataFlux Data Management Studio, SAS Event Stream Processing, and SAS Data Loader for Hadoop.

This reference guide guides you through solutions to address some common EEL tasks. Most examples use the Expression node in the Data Job Editor. All of the examples in this guide also apply to other nodes where EEL is used in DataFlux Data Management Studio.

Introduction to the EEL Operations

Operations in the EEL are processed in symbols. Symbols are similar to variables; they are either fields passed from the node above or are variables declared in the code. EEL code consists of declarations, statements, and labels.

Declarations
: Declarations establish the existence of variables in memory. Declared variables are available only after their declaration, so it is better to make all declarations at the beginning of a code segment. Place declarations in the code outside of programmatic constructs, so declaring a variable in a FOR loop is not valid.

Statements
: Statements are either assignments (for example: x=y) or keywords (for example: goto) followed by parameters. Statements can be located anywhere in a code segment.

Labels
: Labels are named locations in a code segment and can be located anywhere in the code segment. Reserved keywords cannot be used for label names. See Reserved Words.

Pieces of Expression code do not need to be separated by anything, but it is best to use white space and newline characters for readability. Code might include comments. A comment is text within a code segment that is not executed. Comments can be either C style (starts with /* and ends with */) or C++ style (starts with // and continues to the end of a line).

Assume that there are two symbols (output fields from the previous step) named "x" and "y." Here is an example of Expression code:

```
// Declaration of integer z
integer z
// Assignment statement
z=x+y
```

This example creates another symbol (field), "z" and sets the value of z to x + y, making z ready for the next step.

A segment of Expression code can also be a straight expression. In the context of the Expression main code area, if a straight expression value is false, then the row is not sent to output. For example, assume the same fields from the previous example, "x" and "y." Consider the following straight expression in the Expression code area:

```
x<=y
```

EEL in the Expression code area executes on each record of your data set. Only records where the value of x is less than or equal to y are output to the next node. If you have more than one function in the main code area, the last function to execute determines the overall expression value. For example, if the last function returns a true value, then the entire expression returns true.

The following example includes several of the concepts discussed above:

```
// declarations
integer x; /*semicolon is safely ignored, and can use C-style comments*/
```

```
real y

// statements
x=10 y=12.4 /* more than one statement can be on a line */
```

Declaration of Symbols

Declarations have the following syntax:

```
["static"]["private"|"public"]["hidden"|"visible"] type[(*size)] ["array"] identifier
```

where type is:

```
"integer"|"string"|"real"|"boolean"|"date"
```

and identifier is a non-keyword starting with an alphabetic character followed by characters, digits, underscores, or any string delimited by back quotation marks (`). Refer to Reserved Words for a list of reserved keywords.

Note: Size is applicable to the string type only.

Note: The global symbol type is deprecated but is equivalent to static public.

Additional information about declaring symbols:

- The default symbol type is public.
- Private symbols are visible only within the code block in which they are declared.
- Static symbols are public by default. You can also declare a static symbol as private.
- String symbols can be declared with a size. If you assign a value to a string symbol, it is truncated to this size. If you do not specify a size, 255 is used by default.

  Note: The maximum size is 5 MB. However, this applies only to fields within the Expression node. If the symbol is available in the output, it is truncated to 32 KB when the Expression node passes the value on to the next node. For example, if you define a string of length 45 KB, you can work with it inside the expression node. However, it is truncated to 32 KB on output. To override the maximum size, set the EXPRESS_MAX_STRING_LENGTH setting.
- The keyword, bytes, qualifies a string size in bytes. See the previous note for additional details.
- Symbols can be declared anywhere in code except within programmatic constructs, such as loops. It is good practice to declare symbols at the beginning of the code block.
- In the Data Job Editor of Data Management Studio, all symbols declared in code are available in the output unless they are declared private or hidden.
- Before code is executed, symbols are reset to null. If the symbols are declared static or have been declared in the pre-processing step, they retain their value from the previous execution.
- The static keyword can be used when declaring symbols. It specifies that the value of the symbol value is not reset between calls to the expression (between rows read in Data Job Editor). This replaces the global keyword. The pre-processing expression defaults all symbols to static public whether they are declared static or not.
- Hidden and visible keywords can be used when declaring symbols. The default is visible if none is specified. Hidden symbols are not output from the expression step in data jobs. Note that this differs from public and private. Private variables are not

output either, but they are not visible outside the expression block. Hidden variables are visible outside the expression block but are not output.

Public or global symbols declared in one area are available to other areas as follows:

- Symbols declared in Pre-Processing are available to any block.
- Symbols declared in Expression are available to Expression and Post-Processing.
- Symbols declared in Post-Processing are available only to Post-Processing.
- Automatic symbols, which are symbols from the previous step, are available to any of the three blocks.

- To declare a variable with spaces or other special characters in the name, write your variable name between back quotation marks (`). For example:

```
string `my var`
`my var`="Hello"
```

Note: It is the grave accent character (`), also known as the back quote, that is used, and not the apostrophe (') or quotation marks ("). The grave accent is found above the tab key on standard keyboards.

Here are some sample declarations:

```
// a 30-character string available
// only to the code block
private string(30) name

// a 30-byte string
string(30 bytes) name

// a 255-character public string
string address

// a global real number
global real number

// a public date field. Use back
// quotes if symbols include spaces
date `birth date`
```

Statements

Statements have the following syntax:

```
statement:
| "goto" label
| identifier "=" expression
| "return" expression
| "if" expression ["then"] statement ["else" statement]
| "for" identifier ["="] expression ["to"] expression ["step" expression] statement
| "begin" statement [statement...] "end"
| ["call"] function
| "while" expression statement

label: identifier ":"

expression:
described later
```

```
function: identifier "(" parameter [,parameter...] ")"
```

Statements can be separated by a semicolon, a space, or newline character. To group more than one statement together (for example, in a FOR loop), use begin/end.

GOTO and LABEL

GOTO LABEL

LABEL: identifier ":"
A GOTO statement jumps code control to a LABEL statement. A label can occur anywhere in the same code block. For example:

```
integer x
x=0
// label statement called start
start:
     x=x+1
     if x < 10 goto start
```

Assignment

Assigns the value of an expression to a symbol as follows:

- Only read-write symbols can be assigned a value.
- In data jobs, all symbols are read-write.
- A symbol assigned an expression of a different type receives the converted (or coerced) value of that expression. For example, if you assign a number to a string-type symbol, the symbol contains a string representation of that number.
- If the expression cannot be converted into the type of symbol, the symbol's value is null. For example, if you assign a non-date string to a date symbol, it will be set to null.

```
integer num
string str
date dt
boolean b
real r

// assign 1 to num
num=1
// assign Jan 28 '03 to the date symbol
dt=#01/28/03#
// sets boolean to true
b=true
// also sets boolean to true
b='yes'
// sets real to 30.12 (converting from string)
r="30.12"
// sets string to the string representation of the date
str=dt
// sets num to the rounded value of r
num=r
```

Arrays

In the EEL, you can create arrays of primitive types such as integers, strings, reals, dates, and Booleans. It is not possible to create arrays of objects such as dbcursor, dbconnection, regex, and file.

The syntax to create arrays of primitive types is as follows:

- string array string_list
- integer array integer_list
- date array date_list
- Boolean array boolean_list
- real array real_list

There are three supported functions on arrays: DIM, SET, and GET. For more information about arrays, see Arrays.

Return

"**return**" expression

The return statement exits the code block immediately, returning a value.

- In the data jobs, the return type is converted to Boolean.
- If a false value is returned from the expression, the record is not included in the output.

The following is an example of a return statement:

```
// only include rows where ID >= 200
if id < 200
     return false
```

IF/ELSE

"**IF**" expression ["then"] statement ["**ELSE**" statement]

The IF/ELSE statement branches to one or more statements, depending on the expression.

- Use this to execute code conditionally.
- If you need to execute more than one statement in a branch, use begin/end.
- The THEN keyword is optional.
- If you nest IF/ELSE statements, the ELSE statement corresponds to the closest if statement. (See the previous example.) It is better to use begin/end statements if you do this, as it makes the code more readable.

In the following example, you can change the value of age to see different outcomes:

```
string(20) person
integer x
integer y
integer age
Age=10
```

```
if Age < 20 then
     person="child"
else
     person="adult"

if Age==10
     begin
               x=50
          y=20
     end

// nested if/else
if Age <= 60
               if Age < 40
          call print("Under 40")
     // this else corresponds to the inner if statement
     else

          call print("Age 40 to 60")
// this else corresponds to the outer if statement
else
     call print("Over 60")
```

FOR

The FOR loop executes one or more statements multiple times.

- FOR loops are based on a symbol, which is set to some value at the start of the loop, and changes with each iteration of the loop.
- A FOR loop has a start value, an end value, and an optional step value.
- The start, end, and step value can be any expression.
- If the step value is not specified, it defaults to 1.
- If you are starting at a high number and ending at a lower number, you must use a negative step.
- If you need to execute more than one statement in the loop, use begin/end.

For example:

```
integer i
for i = 1 to 10 step 2
call print('Value of i is ' & i)

integer x
integer y
x=10 y=20

for i = x to y
     call print('Value of i is ' & i)

for i = y to x step -1
     begin
          call print('Value of i is ' & i)
          x=i /*does not affect the loop since start/end/step
          expressions are only evaluated before loop*/
          end
```

WHILE

"**WHILE**" expression statement

The WHILE loop enables you to execute the same code multiple times; a condition remains true. For example:

```
integer i
i=1000
// keep looping while the value of i is > 10
while i > 10
     i=i/2

// you can use begin/end to enclose more than one statement
while i < 1000
     begin
          i=i*2
          call print('Value if i is ' & i)
     end
```

BEGIN/END

"**BEGIN**" statement [statement...] "**END**"

The BEGIN/END statement groups multiple statements together. If you need to execute multiple statements in a FOR or WHILE loop or in an IF/THEN/ELSE statement, you must use begin/end. These can be nested as well.

CALL

"**CALL**" statement [statement...] "**END**"

This statement calls a function and discards the return value. By default, the return value of the last function called in an expression is returned, unless there is an explicit expression or "return" statement at the end. This enables you to prevent the function return value from being used to determine the expression if it is the last function in the expression.

Expressions

- An expression can include operators in combination with numbers, strings, functions, functions, which use other functions.
- An expression always has a resulting value.
- The resulting value can be one of the following: string, integer, real, date, and Boolean.
- The resulting value can also be null (a special type of value).

This section covers different types of expressions.

Operators

The following table lists operators in order of precedence:

Operators	Description
! or not	Boolean
(,)	parentheses (can be nested to any depth)
*	multiply
/	divide
%	modulo
+	add
-	subtract
&	string concatenation
!=	not equal ("!=" and "<>" are the same)
<>	not equal
==	comparison operator (= is an assignment and should not be used for comparisons)
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
and	Boolean and
or	Boolean or

Modulo Operator

The modulo operator is represented by the % symbol. The result of the expression **`a%d`** ("**`a modulo d`**") returns a value r, for example:

```
a = qd + r and 0 ≤ r < | d |, where | d | denotes the absolute value of d
```

If either **`a`** or **`d`** are not integers, they are rounded down to the nearest integer before the modulo calculation is performed.

For positive values of **`a`** and **`d`**, it can be the remainder on division of **`a`** by **`d`**. For example:

a	d	a%d (r)
11	3	2
11	-3	2
-11	3	-2
9.4	3	0
9.6	3	0
10	3	1
9.4	3.2	0
9.6	3.2	0
10	3.2	1
–10.2	3.2	–2

Comparison Operator

Do not confuse the comparison operator (==) with the assignment operator (=). For example:

```
// correct statements to compare the value of x and y
if x==y then statement1
else statement2

// Assigning a value
x=y
```

String Expressions

A string expression is a string of undeclared length. Strings can be concatenated using an ampersand (&) or operated upon with built-in functions. For information about defining the length of a string, see Declaration of Symbols. For example:

```
string str
// simple string
str="Hello"
// concatenate two strings
str="Hello" & " There"
```

Note: Setting a string variable to a string expression results in a truncated string if the variable was declared with a shorter length than the expression.

When a string value is used in a Boolean expression, the value is evaluated and the following values are considered true, (upper/lower/mixed): true, t; yes, y; or 1. The following values are considered false (also upper/lower/mixed): false, no, n, or 0.

For more information about string expressions, see Strings.

Integer and Real Expressions

Integer and real expressions result in an integer or real value, respectively. For example:

```
integer x
real r

// order of precedence starts with parentheses,
// then multiplication, then addition
x=1+(2+3)*4

// string is converted to value 10
x=5 + "10"
r=3.14

// x will now be 3
x=r
```

Date Expressions

- A date value is stored as a real value with the whole portion representing number of days since January 1, 1900, and the fraction representing the fraction of a day.
- If years are specified as two digits, then the years between 51 and 99 are assumed to be in 1900. Other years are assumed to be in 2000.
- A date constant is denoted with a number sign (#).
- If a whole number is added to a date, the resulting date is advanced by the specified number of days.
- To make changes to the locale setting in Microsoft Windows, refer to the LOCALE() function topic in this reference guide.

For example:

```
date date_value1
date data_value2
date_value1 = todate("01/02/03")
date_value2 = #01-02-03#
```

Note: The actual results depend on your Windows system settings.

For more on date expressions, see Dates and Times.

Boolean Expressions

- A Boolean expression can either be true or false.
- Results of comparisons are always Boolean.
- Using AND or OR in an expression also results in a Boolean value.

For example:

```
boolean a
boolean b
boolean c

a=true
```

```
b=false
// c is true
c=a or b
// c is false
c=a and b
// c is true
c=10<20
// c is false
c=10==20
// c is true
c=10!=20
// c is true
c='yes'
// c is false
c='no'
```

Null Propagations

If any part of a mathematical expression has a null value, the entire expression is usually null.

The following table shows how nulls are propagated:

Expression	Result
null == value	null (applies to all comparison operators)
null & string	string
null & null	null
number + null	null (applies to all arithmetic operations)
null + null	null (applies to all arithmetic operations)
null AND null	null
null AND true	null
null AND false	false
null OR null	null
null OR true	true
null OR false	false
not null	null
if null	statement following if is not executed
FOR loop	run-time error if any of the terms are null

Expression	Result
while null	statement following while is not executed

For example:

```
integer x
integer y
integer z
boolean b
string s
x=10
y=null
// z has a value of null
z=x + y
// b is true
b=true or null
// b is null
b=false or null
// use isnull function to determine if null
if isnull(b)
     call print("B is null")
// s is "str"
s="str" & null
```

Coercion

If a part of an expression is not the type expected in that context, it is converted into the correct type.

- A type can be coerced into other types.
- If a value cannot be coerced, it results in null.

To explicitly coerce one type to another type, use one of the following functions: TOBOOLEAN, TODATE, TOINTEGER, TOREAL, or TOSTRING. These functions are helpful when there is a need to force a comparison of different types. For example, to compare a string variable called xyz with a number 123.456, the number is converted to a string before the comparison is completed using the following example:

```
toreal(xyz) > 123.456
```

The following table shows the rules for coercion:

Coercion Type	TOSTRING	TOINTEGER	TOREAL	TODATE	TOBOOLEAN
from String		yes	yes	yes	yes
from Integer	yes		yes	yes	yes
from Real	yes	yes		yes	yes
from Date	yes	yes	yes		no
from Boolean	yes	yes	yes	no	

The following table shows special considerations for coercion:

Coercion Type	Resulting Action
date to string	A default date format is used: YYYY/MM/DD hh:mm:ss. Use the FORMATDATE function for a more flexible conversion.
date to number	The number represents days since 12/30/1899. Hours, minutes, seconds, and milliseconds are converted to a fraction, where 1 hour = 1/24 units, 1 minute = 1/(24*60) units, and so on.
string to date	Most date formats are recognized and intelligently converted. See the Date Expressions section for more information.
string to Boolean	The values yes, no, true, false, y, n, t, and f are recognized.
integer to real to Boolean	Any nonzero value is true. Zero is false.

Functions

- A function can be part of an expression.
- If you need to call a function but do not want the return value, use CALL.
- Each function has a specific return type and parameter type.
- If the parameters provided to the function are not the correct type, they are sometimes coerced.
- A function sometimes requires a parameter to be a specific type. If you pass a parameter of the wrong type, it is not coerced and you get an error.
- Functions normally propagate null (there might be exceptions).
- Some functions might modify the value of their parameters if they are documented to do so.
- Some functions might accept a variable number of parameters.

For example:

```
string str
integer x
str="Hello there"
// calls the upper function
if upper(str)=='HELLO THERE'
     // calls the print function
     call print("yes")

// x is set to 7 (position of word 'there')
x=instr(str,"there",1)
```

Global Functions

You can register global functions (or user-defined functions, known as UDFs) that can be reused from any expression in the system. This includes data and process flow nodes, business rules, and more. To do this, create one text file (or more) in the installation under etc/udf. Each file might contain one or more function definitions. Each definition is enclosed in a function/end function block. The return type for each function can be integer, real, date, string, or Boolean. To get to function parameters, use the PARAMETER() and PARAMETERCOUNT() functions as well as individual functions for the types (PARAMETERBOOLEAN, PARAMETERDATE, PARAMETERINTEGER, PARAMETERREAL, or PARAMETERSTRING).

The following functions are applicable when registering the global functions mentioned earlier. Each global function accepts an integer as a parameter to indicate which parameter is desired. Each function returns the type specified and converts the parameter to that type, if it is not already the type specified.

- PARAMETERBOOLEAN
- PARAMETERDATE
- PARAMETERINTEGER
- PARAMETERREAL
- PARAMETERSTRING

For example:

```
function example_udf return string 255
   // this function is declared to return a string of
   // up to 255 characters retrieve the number of
   // parameters passed
   print("You passed " & parametercount() & " parameters")
   integer x
   for x = 1 to parametercount()
   begin
      print("Parameter " & x & " type is " &
      parametertype(x) & " value is " & parameter(x))
      if(parametertype(x)=="integer")
         print("Integer: " & parameterinteger(x))
      else if parametertype(x)=="real"
         print("Real: " & parameterreal(x))
   end
   return "string val"
end function
```

Objects

The EEL supports a number of objects. Generally, an object is a type of code in which not only the data type of a data structure is defined, but also the types of operations that can be applied to the data structure. In particular, the EEL supports objects for the following:

- Data Quality: Expressions and Functions
- Databases: Database connectivity (DBCONNECT object)
- Files: Text file reading and writing (FILE object)

- Regular Expressions: Regular expression searches (REGEX object)

Chapter 2
Data Job Expressions Node

Data Job Expressions Node

The DataFlux Data Management Studio Expressions node is a utility that enables you to create your own nodes using the Expression Engine Language (EEL) scripting language.

For information about the Data Job Expressions node, refer to the DataFlux Data Management Studio online Help.

Part 2

Expression Engine Language Functions

Chapter 3

Array Functions

Array Functions

In the Expression Engine Language (EEL), it is possible to create arrays of simple types such as string, integer, date, Boolean, and real. Currently there are three functions that apply to array types: DIM, GET, and SET.

Dictionary

DIM Function

Creates, resizes, or determines the size of an array. If a parameter is specified, the array is resized or created. The new size is returned.

Category:	Array
Returned data type:	Integer

Syntax

arrayName.**DIM**<(*newsize*)>

Required Argument

arrayName
is the name of the array that you declared earlier in the process.

Optional Argument

newsize
is the optional numeric size (dimension) of the array. This can be specified as a numeric constant, field name, or expression.

Details

The DIM function is used to size and resize the array. It creates, resizes, or determines the size of the array. If a parameter is specified, the array is created or resized. The supported array types include:

- String
- Integer
- Date
- Boolean
- Real

Example

```
// declare the string array
string array string_list
// Set the dimension of the String_List array to a size of 5
rc = string_list.dim(5) // outputs 5
// <omitted code to perform some actions on the array>
// Re-size the array size to 10
rc = string_list.dim(10) // outputs 10
// Query the current size
re = string_list.dim() // outputs 10
```

GET Function

Retrieves the value of the specified item within an array. The returned value is the value of the array.

Category:	Array
Returned data type:	Integer

Syntax

array name.**GET**(<*n*>)

Required Argument

array name
is the name of the array that you declared earlier in the process.

Optional Argument

n
is the index of the array element for which the content is retrieved. This can be specified as a numeric constant, field name, or expression.

Details

The GET function returns the value of a particular element in the array.

Examples

Example 1

```
// Declare the string array "string_list" and Integer "i"
string array string_list
integer i

// Set the dimension of string_list array to 5 and initialize the counter (i) to 1
string_list.dim(5)
i=1

// Set and print each entry in the array, incrementing the counter by 1
while(i<=5)
begin
   string_list.set(i,"Hello")
   print(string_list.get(i))
   i=i+1
end
```

Example 2

```
string array string_list
integer i

// set the dimension
string_list.dim(5)
i=1

// set and print each entry in the array
while(i<=5)
begin
     string_list.set(i,"Hello")
     print(string_list.get(i))
     i=i+1
end

// resize the array to 10
string_list.dim(10)
while(i<=10)
begin
     string_list.set(i,"Goodbye")
     print(string_list.get(i))
     i=i+1
end
```

SET Function

Sets values for items within an array. The returned value is the old value of the specified element in the array.

Category: Array

Returned data type: Integer

Syntax

array name.**SET**(<*n*,*"string"*>)

Required Argument

array name
is the name of the array that you declared earlier in the process.

Optional Arguments

n
is the number of the dimension that you are setting the value for; this can be specified as a numeric constant, field name, or expression.

"string"
is the value that you want to place into the array element; this can be specified as a string constant, field name, or expression.

Details

The SET function sets the value of an entry in the array.

Examples

Example 1

```
// Declare the string array "string_list"
// Set the dimension of string_list array to 5
string array string_list
string_list.dim(5)

// Set the first string element in the array to "Hello"
string_list.set(1,"Hello")
```

Example 2

```
string array string_list
string_list.dim(5)
// sets the first string element in the array to hello
string_list.set(1,"hello")
```

Chapter 4

Data Quality Functions

Data Quality Functions

Expression Engine Language (EEL) supports the data quality object. You can use data quality to perform the listed functions (object methods) from within the EEL node. Some of the advantages of using data quality functions within the EEL include dynamically changing match definitions, reading match definitions from another column, or setting different definitions.

Dictionary

DQ.CASE Function

Applies casing rules (upper, lower, or proper) to a string. The function also applies context-specific casing logic using a case definition in the SAS Quality Knowledge Base (QKB).

Category: Data Quality

Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.CASE(case_def, casing_type, input, result)

Required Arguments

casing_type
integer numeric constant that specifies the type of casing that is applied, [1 = uppercase, 2 = lowercase, 3 = proper case].

input
a string representing the input value or input field name.

result
a string representing the output field name.

Optional Argument

case_def
a string representing the name of a case definition in the QKB. Pass an empty string to use the default casing algorithm.

Details

The DQ.CASE function applies casing rules to an input string and outputs the result to a field.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ_INITIALIZE.

You can specify one of three casing types: uppercase, lowercase, or propercase. When uppercase or lowercase is specified, the function applies Unicode uppercase or lowercase mappings to the characters in the input string. When propercasing is specified, the function applies uppercase mappings to the first letter in each word and lowercase mappings to the remaining letters.

The caller can invoke the use of a case definition. A case definition is an object in the QKB that contains context-specific casing logic. For example, a case definition implemented for the purpose of propercasing name data can be used to convert the string "Mcdonald" to "McDonald". Refer to the QKB documentation for information about what case definitions are available in your QKB. If you do not want to use a case definition, you can omit the case definition name by entering a blank string for the case definition parameter. In this case, generic Unicode case mappings are applied to the input string as described earlier.

Note: If you want to use a case definition, you must call DQ.LOADQKB before calling DQ.CASE. The function DQ.LOADQKB loads the contents of a QKB into memory and links that QKB with the data quality object. This enables DQ.CASE to access the case definition that you specify.

Example

```
data quality dq
```

```
 string output
 dq = dq_initialize()
 dq.case("", 1, "ronald mcdonald", output)
 // outputs "RONALD MCDONALD"

dq.case("", 3, "ronald mcdonald", output)
 // outputs "Ronald Mcdonald"

dq.loadqkb("ENUSA")
 dq.case("Proper (Name)", 3, "ronald mcdonald", output)
 // outputs "Ronald McDonald"
```

DQ.EXTRACT Function

Extracts attributes from a string.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is a value, token, or token value from the extract function.

Syntax

DQ.EXTRACT*definition, string*

Required Arguments

definition
a string representing the name of the extraction definition in the QKB.

string
a string that represents the attribute that needs to be extracted.

Details

The DQ.EXTRACT function extracts attributes from a string into tokens. The first parameter is the name of the QKB extraction definition. The second is the string where the attributes are extracted. This function returns a number of tokens that were created. It returns 0 if it fails.

Example

```
data quality dq
string output
integer o
integer i

/* Initialize DQ */
dq = dq_initialize()
dq.loadqkb("EN")

/* Extract using the "Product Attributes" Extraction definition (using QKB PD 2012A) */
o = dq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")
```

```
/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

DQ.GENDER Function

Determines the gender of an individual's name using a gender analysis definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.GENDER(*gender_def, input, result*)

Required Arguments

gender_def
a string representing the name of a gender analysis definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.GENDER function analyzes a string representing an individual's name and determines the gender of the name.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.GENDER, you must specify the name of a gender analysis definition. A gender analysis definition is an object in the QKB that contains reference data and logic used to determine the gender of the input name string. See your QKB documentation for information about which gender analysis definitions are available in your QKB.

Example

```
data quality dq
 string output
dq = dq_initialize()
dq.loadqkb("ENUSA")
dq.gender("Name", "John Smith", output)
 // outputs "M"

 dq.gender("Name", "Jane Smith", output)
 // outputs "F"

 dq.gender("Name", "J. Smith", output)
 // outputs "U" (unknown)
```

DQ.GETLASTERROR Function

Returns a string describing the most recent error encountered by a data quality object.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is a string containing an error message.

Syntax

DQ.GETLASTERROR(<>)

Details

The DQ.GETLASTERROR function is a member of the data quality class. It returns an error message describing the most recent error encountered by a data quality object. The error might have occurred during invocation of any other data quality member function.

A best practice for programmers is to check the result code for each data quality call. If a result code indicates failure, use DQ.GETLASTERROR to retrieve the associated error message.

Example

```
data quality dq
 integer rc
 string errmsg
dq = dq_initialize()
 rc = dq.loadqkb("XXXXX")
 // an invalid locale name -- this will cause an error

 if (rc == 0) then
      errmsg = dq.getlasterror()
 // returns an error message
```

DQ.IDENTIFY Function

Identifies the context of a string using an identification analysis definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.IDENTIFY(*ident_def, input, result*)

Required Arguments

ident_def
a string representing the name of an identification analysis definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.IDENTIFY function analyzes a string and determines the context of the string. The context refers to a logical type of data, such as name, address, or phone.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized using a call to the function DQ.INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.IDENTIFY, you must specify the name of an identification analysis definition. An identification analysis definition is an object in the QKB that contains reference data and logic used to identify the context of the input string. Refer to your QKB documentation for information about which identification analysis definitions are available in your QKB.

Note: For each identification analysis definition, there is a small set of possible contexts that might be output. Refer to the description of an identification analysis definition in the QKB documentation to see which contexts that definition is able to identify.

Example

```
data quality dq
 string output
 dq = dq_initialize()
dq.loadqkb("ENUSA")
dq.identify("Individual/Organization", "John Smith", output)
 // outputs "INDIVIDUAL"
```

```
dq.identify("Individual/Organization", "DataFlux Corp", output)
 // outputs "ORGANIZATION"
```

DQ_INITIALIZE Function

Instantiates and initializes a data quality object.

Category:	Data Quality
Returned data type:	Character
Note:	The returned value is an initialized instance of a data quality object.

Syntax

DQ_INITIALIZE(<>)

Details

The DQ_INITIALIZE instantiates and initializes a data quality object. The object can then be used to invoke data quality class functions.

Example

```
data quality dq
dq = dq_initialize()
```

DQ.LOADQKB Function

Loads definitions from a QKB into memory and links those definitions with the data quality object.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.LOADQKB(*locale*)

Required Argument

locale
a five-character locale code name representing a locale supported by the QKB.

Details

The function DQ.LOADQKB is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The function DQ.LOADQKB can be called after the initialization.

The DQ.LOADQKB function loads definitions from a QKB into memory and links those definitions with the data quality object. A definition is a callable object that uses context-sensitive logic and reference data to perform analysis and transformation of strings. Definitions are used as parameters in other dq functions.

When calling DQ.LOADQKB, you must specify a locale code. This locale code is a five-character string representing the ISO codes for the locale's language and country. Refer to your QKB documentation for a list of codes for locales that are supported in your QKB.

Note: Only one locale code can be specified in each call to DQ.LOADQKB. Only definitions associated with that locale are loaded into memory. This means that support for only one locale at a time can be loaded for use by a data quality object. In order to use QKB definitions for more than one locale, you must either use multiple instances of the data quality class or call DQ.LOADQKB multiple times for the same instance, specifying a different locale with each call.

Example

```
data quality dq_en
 // we instantiate two dq objects

data quality dq_fr
 string output_en
 string output_fr
dq_en = dq_initialize()
dq_fr = dq_initialize()

 dq_en.loadqkb("ENUSA"
 // loads QKB support for locale English, US

 dq_fr.loadqkb("FRFRA")
 // loads QKB support for locale French, France

 dq_en.gender("Name", "Jean LaFleur", output_en)
 // output is 'U'

dq_fr.gender("Name", "Jean LaFleur", output_fr)
 // output is 'M'
```

DQ.MATCHCODE Function

Generates a match code for a string using a match definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.MATCHCODE(*match_def, sensitivity, input, result*)

Required Arguments

match_def
a string representing the name of a match definition in the QKB.

sensitivity
integer numeric constant that specifies the sensitivity level to be used when generating the match code [possible values are 50–95].

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.MATCHCODE function generates a match code for an input string and outputs the match code to a field. The match code is a fuzzy representation of the input string. It can be used to do a fuzzy comparison of the input string to another string.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.MATCHCODE, you must specify the name of a match definition. A match definition is an object in the QKB that contains context-specific reference data and logic used to generate a match code for the input string. Refer to your QKB documentation for information about which match definitions are available in your QKB.

You must also specify a level of sensitivity. The sensitivity indicates the level of fuzziness that is used when generating the match code. A higher sensitivity means that the match code is less fuzzy (yielding fewer false positives and more false negatives in comparisons). A lower sensitivity means that the match code is more fuzzy (yielding fewer false negatives and more false positives in comparisons). The valid range for the sensitivity parameter is 50–95.

Example

```
data quality dq
 string output
dq = dq_initialize()
 dq.loadqkb("ENUSA")
 dq.matchcode("Name", 85, "John Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"

dq.matchcode("Name", 85, "Johnny Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"
```

DQ.MATCHSCORE Function

Processes two input strings along with the name of the match definition and outputs the sensitivity for the match strings.

Category: Data Quality

Syntax

DQ.MATCHSCORE(*definition_name, input1, input2, use_wildcards*)

Required Arguments

definition_name
the name of the match definition to use.

input1
the first input string to check.

input2
the second input string to check.

returns
the sensitivity value.

use_wildcards
true if wildcards in generated match codes should be considered for the purpose of scoring.

Details

The DQ.MATCHSCORE function determines the highest sensitivity value where two input strings generate the same match code.

Example

```
data quality dq;
dq = dq_initialize();
dq.loadqkb("ENUSA");

integer x;
x = dq.matchscore("Name", "John Jones", "John J. Jones", false);
```

DQ.PARSE Function

Parses a string.

Category: Data Quality

Syntax

DQ.PARSE(*definition, string*)

Required Arguments

definition
a string representing the parsed data.

returns
the number of tokens.

string
the input string to be parsed into tokens.

Details

The DQ.PARSE function parses the input string into tokens. The first parameter is the name of the QKB parse definition. The second parameter is the string from which the tokens are parsed. This returns the number of tokens created. It returns 0 if it fails.

Example

```
/* Parse (using QKB CI 2013A) */
o = dq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.PATTERN Function

Generates a pattern for a string using a pattern analysis definition in the QKB.

Category: Data Quality

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.PATTERN(*pattern_def, input, result*)

Required Arguments

pattern_def
a string representing the name of a match definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.PATTERN function generates a pattern for the input string and outputs the pattern to a field. The pattern is a simple representation of the characters in the input string. Such patterns can be used to perform pattern frequency analysis for a set of text strings.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.PATTERN, you must specify the name of a pattern analysis definition. A pattern analysis definition is an object in the QKB that contains logic used to generate a pattern for the input string. Refer to your QKB documentation for information about which pattern analysis definitions are available in your QKB.

Example

```
data quality dq
 string output
dq = dq_initialize()
 dq.loadqkb("ENUSA")
 dq.pattern("Character", "abc123", output)
 // Outputs "aaa999"
```

DQ.STANDARDIZE Function

Generates a standard for a string using a standardization definition in the QKB.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DQ.STANDARDIZE(*stand_def, input, result*)

Required Arguments

stand_def
a string representing the name of a standardization definition in the QKB.

input
a string representing the input value or input field name.

result
a string representing the output field name.

Details

The DQ.STANDARDIZE function generates a normalized standard for an input string and outputs the standard to a field.

The function is a member of the data quality class. A data quality object can be declared as a variable and must then be initialized through a call to the function DQ_INITIALIZE. The member function DQ.LOADQKB must then be called to load the contents of a QKB into memory and link that QKB with the data quality object. The data quality object then retains information about the QKB locale setting and the QKB locale setting.

When calling DQ.STANDARDIZE, you must specify the name of a standardization definition. A standardization definition is an object in the QKB that contains context-specific reference data and logic used to generate a standard for the input string. Refer to your QKB documentation for information about which standardization definitions are available in your QKB.

Example

```
data quality dq
 string output
 dq = dq_initialize()
dq.loadqkb("ENUSA")
dq.standardize("Name", "mcdonald, mister ronald", output)
 // Outputs "Mr Ronald McDonald"
```

DQ.TOKEN Function

Obtains a token name for the index from a parse or extract function.

Category: Data Quality

Returned data type: Character

Note: This function returns true on success and false if the index is out of range. The token name is returned in the second parameter from the parse or extract function.

Syntax

DQ.TOKEN(integer, string)

Required Arguments

integer
the index of the token for which the name is desired.

string
the output string that receives the token name.

Details

The DQ.TOKEN function is used to retrieve an extraction or parse token name for the index. This function follows a parse or extract function.

Examples

Example 1

```
data quality dq
string output
integer o
integer i

/* Initialize DQ */
dq = dq_initialize()
dq.loadqkb("EN")

/* Extract using the "Product Attributes" Extraction definition (using QKB PD 2012A) */
o = dq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.TOKENVALUE Function

Obtains an attribute from the last parse or extract function.

Category:	Data Quality
Returned data type:	Character

Note: The returned value is a token value from the parse or extract function. Returns true if the token value is found or false if not.

Syntax

DQ.TOKENVALUE(<*token string, output string*>)

Required Arguments

token string
returns true if the token is found and false if the token is not found.

output string
a string that represents the value for that token.

Details

The DQ.TOKENVALUE function is used to retrieve an extraction or parse result value. The first parameter is the token name. It returns the attribute stored in that token in the second parameter. It returns true on success and false on failure (for example, if the token was not found). This function follows a parse or extract function.

Examples

Example 1

```
data quality dq
string output
integer o
integer i

/* Initialize DQ */
dq = dq_initialize()
dq.loadqkb("EN")

/* Extract using the "Product Attributes" Extraction definition (using QKB PD 2012A) */
o = dq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dq.parse("Name", "Mr. John Q Public Sr")
```

```
/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.VALUE Function

Retrieves the value from the last parse or extract function.

Category:	Data Quality
Returned data type:	Integer
Note:	The returned value is true if the index is valid.

Syntax

DQ.VALUE(*integer, string*)

Required Arguments

integer
represents an index of the token.

string
a string that represents the output value.

Details

The DQ.VALUE function is used to retrieve an extraction or parse result. The first parameter is the index of a token. The second parameter receives the attribute that is stored in the token. The function returns true if it is able to get the token value. It returns false if it fails. This function follows a parse or extract function.

Examples

Example 1

```
data quality dq
string output
integer o
integer i

/* Initialize DQ */
dq = dq_initialize()
```

```
dq.loadqkb("EN")

/* Extract using the "Product Attributes" Extraction definition (using QKB PD 2012A) */
o = dq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

Example 2

```
/* Parse (using QKB CI 2013A) */
o = dq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
     dq.token(i, output)
     print ("token #" & i & " = " & output)
     dq.value(i, output)
     print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

Chapter 5

Database Functions

Overview

To work with databases, use the DBCONNECT object in Expression Engine Language (EEL). You can connect to data sources using built-in functions that are associated with the DBCONNECT object. You can also return a list of data sources, and evaluate data input from parent nodes.

Overview of the Database Objects

The database objects enable you to use the EEL to connect directly to a relational database system and execute commands on that system as part of your expression code. There are three objects associated with this functionality:

DBCONNECTION
a connection to the database

DBSTATEMENT
a prepared statement

DBCURSOR
a cursor for reading a result set

Releasing Database Objects

When objects are set to null, they are released. Depending on whether objects are defined as static or nonstatic, see Declaration of Symbols for additional details. When symbols are not automatically reset to null, you need to use the release() methods to explicitly release database objects.

Dictionary

DBCONNECT Function

Uses the EEL to connect directly to a relational database system and execute commands on that system as part of your expression code. DBCONNECT is a global function.

Category: Database

Syntax

DBCONNECT(<*"connect_string"*>)

Required Arguments

returns
a DBCONNECTION object

"connect_string"
a string containing the name and path to connect to the database.

Optional Argument

share id
allows you to share connections between database and expression nodes. If the same share id is specified then it will cause any connection with that share id to use the same database connection.

DBDATASOURCES Function

Returns a list of data sources as a dbcursor. DBDATASOURCES is a global function.

Category: Database

Syntax

DBDATASOURCES(< >)

Without Arguments

Returns a list of data sources as a dbcursor. The data source includes:

NAME
a string containing the name of the data source.

DESCRIPTION
a string containing the driver (shown in the ODBC Administrator and DataFlux Connection Administrator).

Type
: an integer containing the subsystem type of the connection [1 = ODBC; 2 = DataFlux TKTS].

HAS_CREDENTIALS
: a Boolean representing if save connection exists.

USER_DESCRIPTION
: a string containing the user-defined description of the data source (defined in the ODBC Administrator and DataFlux Connection Administrator).

Example

```
// list all data sources
dbconnection db_conn
db_conn=dbconnect("dsn=dsn name")
dbcursor db_conn.db_curs
db_curs = dbdatasources()
ncols = db_curs.columns()

while db_curs.next()
begin
   for i_col=0 to ncols-1
   begin
      colname = db_curs.columnname(i_col)
      coltype = db_curs.columntype(i_col)
      collength = db_curs.columnlength(i_col)
      colvalue = db_curs.valuestring(i_col)
      pushrow()
   end
end

db_curs.release()
db_conn.release()
```

Chapter 6

Date and Time Functions

Overview

Dates, along with integers, reals, Booleans, and strings, are considered basic data types in the EEL. Similar to other basic data types, EEL provides functions to perform operations on dates.

Dictionary

FORMATDATE Function

Returns a date formatted as a string.

Category:	Date and Time
Returned data type:	String
Note:	The returned value is a string with the date formatted as a string.

Syntax

FORMATDATE(<datevar,format>)

Required Arguments

<datevar>
a date that needs to be formatted; this can be specified as field name.

<format>
a string that represents the format that needs to be applied; this can be specified as fixed string, field name, or expression.

Details

The format parameter can include any string, but the following strings are replaced with the specified values:

- YYYY: four-digit year
- YY: two-digit year
- MMMM: full month in proper case
- MMM: abbreviated three-letter month
- MM: two-digit month
- DD: two-digit day
- hh: hour
- mm: minute
- ss: second
- ms: millisecond

Note: The format parameters are case sensitive.

The FORMATDATE function dates should be in the format specified by ISO 8601 (YYYY-MM-DD hh:mm:ss:ms) to avoid ambiguity. Remember that date constants must start with and end with the # sign (for example, #12-February-2010#).

Examples

Example 1

```
// Declare a date variable and initialize
 // it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dteval, "MM/DD/YY")
Results: 02/12/10
```

Example 2

```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval, "DD MMM YYYY")
Results: 12 Feb 2010
```

Example 3

```
// Declare a date variable and initialize
// it to a value
```

```
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval, "MMMM DD, YYYY")
Results: February 12, 2010
```

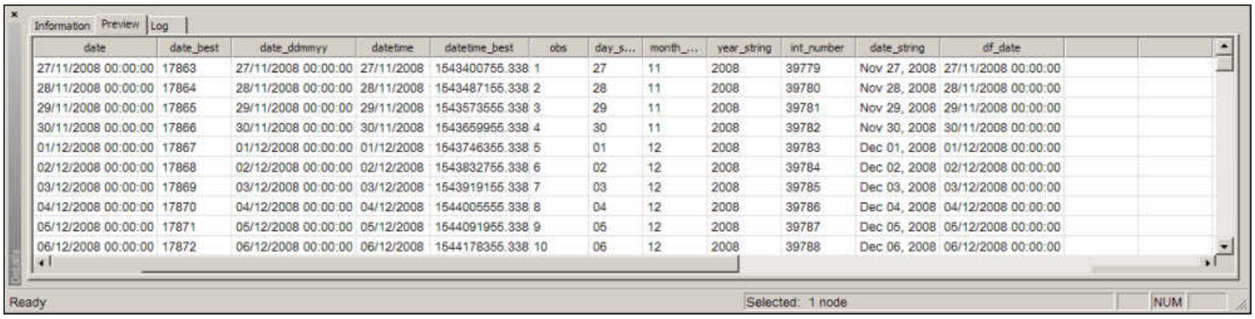
Example 4

```
day_string=formatdate('date',"DD");
month_string=formatdate('date',"MM");
year_string=formatdate('date',"YYYY");

int_number='date';
date_string=formatdate(int_number,"MMM DD,YYYY");

df_date='date';
```

Figure 6.1 *Results*



date	date_best	date_ddmmyy	datetime	datetime_best	obs	day_s...	month_...	year_string	int_number	date_string	df_date
27/11/2008 00:00:00	17863	27/11/2008 00:00:00	27/11/2008	1543400755.338	1	27	11	2008	39779	Nov 27, 2008	27/11/2008 00:00:00
28/11/2008 00:00:00	17864	28/11/2008 00:00:00	28/11/2008	1543487155.338	2	28	11	2008	39780	Nov 28, 2008	28/11/2008 00:00:00
29/11/2008 00:00:00	17865	29/11/2008 00:00:00	29/11/2008	1543573555.338	3	29	11	2008	39781	Nov 29, 2008	29/11/2008 00:00:00
30/11/2008 00:00:00	17866	30/11/2008 00:00:00	30/11/2008	1543659955.338	4	30	11	2008	39782	Nov 30, 2008	30/11/2008 00:00:00
01/12/2008 00:00:00	17867	01/12/2008 00:00:00	01/12/2008	1543746355.338	5	01	12	2008	39783	Dec 01, 2008	01/12/2008 00:00:00
02/12/2008 00:00:00	17868	02/12/2008 00:00:00	02/12/2008	1543832755.338	6	02	12	2008	39784	Dec 02, 2008	02/12/2008 00:00:00
03/12/2008 00:00:00	17869	03/12/2008 00:00:00	03/12/2008	1543919155.338	7	03	12	2008	39785	Dec 03, 2008	03/12/2008 00:00:00
04/12/2008 00:00:00	17870	04/12/2008 00:00:00	04/12/2008	1544005555.338	8	04	12	2008	39786	Dec 04, 2008	04/12/2008 00:00:00
05/12/2008 00:00:00	17871	05/12/2008 00:00:00	05/12/2008	1544091955.338	9	05	12	2008	39787	Dec 05, 2008	05/12/2008 00:00:00
06/12/2008 00:00:00	17872	06/12/2008 00:00:00	06/12/2008	1544178355.338	10	06	12	2008	39788	Dec 06, 2008	06/12/2008 00:00:00

TODAY Function

Returns the current data and time. This function is based on the local time zone.

Category:	Date and Time
Returned data type:	Character
Note:	The returned value is a date that represents the current date and time value.

Syntax

TODAY(< >)

Details

The TODAY function returns the current date and time value. For example, at 4:00 p.m. on February 12, 2010, the function would return the value "02/12/10 4:00:00 PM". Although it is represented as a character string, the actual value is a date value. For more information, see Date Expressions.

Before using this function, you must first declare a date variable to contain the date/time value.

Example

```
// declare the date variable to contain the date and time
date currentdate
// Use the TODAY function to populate the date variable with the current date and time
currentdate = TODAY()
```

TODAYGMT Function

Returns the current date and time. This function is based on Greenwich Mean Time (GMT).

Category:	Date and Time
Returned data type:	Character
Note:	The returned value is a date and time combination that represents the current GMT date and time value.

Syntax

TODAYGMT(< >)

Details

The TODAYGMT function returns the current GMT date and time value. For example, at 4:00 p.m. on February 12, 2010, the function would return the value "02/12/10 4:00:00 PM". Although it is represented as a character string, the actual value is a date value.

Before using this function, you must first declare a date variable to contain the date/time value.

Example

```
// declare the date variable to contain the date and time
date currentdate
// Use the today function to populate the date variable with the current date and time
currentdate = todaygmt()
```

Chapter 7

Execution Functions

Overview

The following execution functions are available for the EEL:

- PUSHROW
- SETOUTPUTSLOT
- SLEEP

Dictionary

PUSHROW Function

Pushes the current values of all symbols (this includes both field values for the current row and defined symbols in the code) to a stack.

Category:	Execution
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

PUSHROW(<Boolean >)

Optional Argument

reprocess
: if true, then it causes the row to be processed again through the expression. If not specified, the row is not processed. again

Details

The PUSHROW function pushes the current values of all symbols (this includes both field values for the current row and defined symbols in the code) to a stack. When the next row is requested, it is given from the top of the stack instead of being read from the step above. When a row is given from the stack, it is not to be processed through the expression again unless the reprocess parameter is specified. When the stack is empty, the rows are read from above as usual. This function always returns true.

Example

```
string name
integer age
date Birthday

name="Mary"
age=28
Birthday=#21/03/1979#

pushrow()

name="Joe"
age=30
Birthday=#21/03/1977#
pushrow()
```

SETOUTPUTSLOT Function

Sets the output slot to slot. This becomes the output slot when the expression exits.

Category:	Execution
Returned data type:	Integer

Syntax

SETOUTPUTSLOT(integer)

Required Arguments

integer
: the slot number for the node that should execute next.

returns
: Boolean [true = success; false = error]

slot
: an integer representing the active output slot when the expression node exits.

Details

Note: The SETOUTPUTSLOT function is applicable in a process job.

The SETOUTPUTSLOT function tells the node (the expression node in which it is running) to exit on the specified slot. In a process job, if you follow a node by two other nodes, you specify a slot for each for example, 0 and 1. If you run SETOUTPUTSLOT(1), it tells the process job to continue with the node that is linked at slot 1. It SETOUTPUTSLOT is not called, it exits on 0 by default.

Example

The following statements illustrate the SETOUTPUTSLOT function.

```
if tointeger(counter)<= 5 then SETOUTPUTSLOT(0)
else SETOUTPUTSLOT(1)
```

SLEEP Function

Pauses execution for the specified number of milliseconds. It allows the job to be interrupted or canceled without problems.

Category:	Execution
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

SLEEP(duration)

Required Argument

<duration>
an integer representing the number of milliseconds to pause

Details

The SLEEP function causes the job to pause for the specified number of milliseconds.

Example

```
sleep(000)
```

Results: The job sleeps for 5 seconds

Chapter 8
File Functions

Overview

Use the external file object to work with files in the Expression Engine Language (EEL). Read and Write operations are supported in the file object and there are additional functions for manipulating and working with files.

Overview of the File Object

The file object can be used to open, read, and write files. A file is opened using the file object. For example:

```
File f
f.open("c:\filename.txt","r")
```

In this example, the OPEN() function opens filename.txt. The mode for opening the file is read. Other modes are "a" (append to end of file) and "w" (write). A combination of these switches can be used.

Executing Programs and File Commands

To execute programs, use the EXECUTE() function:

```
execute(string)
```

For example, the following code changes the default permissions of a text file created by the Data Job Editor.

To execute the command in Microsoft Windows, enter:

```
execute("/bin/chmod", "777", "file.txt")
```

Or, to execute from the UNIX shell, enter:

```
execute("/bin/sh", "-c", "chmod 777 file.txt")
```

Running a Batch File By Using Execute Function

To invoke the MS-DOS command prompt, call the cmd.exe file. For example:

```
//Expression
execute("cmd.exe", "/q" ,"/c", C:\BatchJobs.bat");
```

The following parameters can be declared:

- /q — turns echo off
- /c — Executes the specified command for the MS-DOS prompt and then closes the prompt

Note: The expression engine handles the backslash character differently; it does not need to be escaped.

For example: "C:\\Program Files\\DataFlux" should now be entered as "C:\Program Files \DataFlux"

Dictionary

CLOSE Function

Closes an open file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**CLOSE**

Details

The CLOSE method closes the file that is currently open file (which was opened by using a fileobject.OPEN call) is closed.

Example

```
file myfile
if ( myfile.open("data.txt") ) then ...
rc = myfile.close()
```

COPYFILE Function

Copies a file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

COPYFILE(<source_file, target_file>)

Required Arguments

source_file
a string representing the name of the file to be copied; this can be specified as a fixed string, field name, or expression.

target_file
a string representing the name of the file to be written; this can be specified as a fixed string, field name, or expression.

Details

The COPYFILE function copies a file. If the target file exists, the COPYFILE function overwrites the target file.

Example

```
string source_file
string target_file
boolean rc_ok

source_file="C:\mydata.txt"
target_file="C:\mydata_copy.txt"

rc_ok = copyfile(source_file, target_file)
```

DELETEFILE Function

Deletes the specified file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

DELETEFILE(<filename>)

Required Argument

filename
a string representing the name of the file to be deleted; this can be specified as a fixed string, field name, or expression.

Details

The DELETEFILE function deletes a file from disk. If the file did not exist, then the return code will be set to false.

Example

```
string filename
boolean rc_ok

filename="C:\mydata_copy.txt"

rc_ok = deletefile(filename)
```

EXECUTE Function

Runs the specified program.

Category:	External File
Returned data type:	Integer
Note:	The returned value represents the existing status of the program. If an error occurs, such as the program not found, then -1 is returned.

Syntax

EXECUTE(<filename<option1, option2,..., <option N>>

Required Argument

filename
a string representing the file (or command) to be executed; this can be specified as a fixed string, field name, or expression.

Optional Argument

option1...N
[optional] a string representing options that are passed to the file (command) that is going to be executed; this can be specified as a fixed string, field name, or expression.

Details

The execute function invokes a file (or operating system command).

Note: Use single quotation marks for any parameter with embedded spaces. For example:

```
execute('cmd.exe','/C',
     '"C:\Program Files (x86)\DataFlux\DMStudio\studio24\bin\dmpexec.cmd"',
     '-j', '"C:\Repository24\batch_jobs\my data job.ddf"',
     '-l', '"C:\temp\log my data job.log"')
```

Example

```
integer rc

// Windows example
rc = execute("cmd.exe", "/Q", "/C", "C:\mybatchjob.bat")
// /Q turns echo off
// /C executes the command specified by filename and then closes the prompt

// Unix example
rc = execute("/bin/sh", "-c", "chmod 777 file.txt")
```

FILEDATE Function

Checks the creation or modification date of a file.

Category:	External File
Returned data type:	Character
Note:	The returned value is a date.

Syntax

FILEDATE(<*filename,*>< *datetype*>)

Required Argument

filename
a string representing the name of the file for which the creation or modification date needs to be retrieved; this can be specified as a fixed string, field name, or expression.

Optional Argument

datetype
a Boolean that specifies whether the creation date or the modification date needs to be returned; this can be specified as a fixed string, field name, or expression [true = modification date, false = creation date].

Details

The FILEDATE function returns either the creation date or the most recent modification date of a file. If the file does not exist a (null) value is returned. If the argument DATETYPE is omitted, the function behaves like the value would have been specified as false.

Example

```
string filename
date creation_date
date modification_date

filename="C:\mydata.txt"

modification_date = filedate(filename, true)
creation_date = filedate(filename, false)
```

FILEEXISTS Function

Checks whether a specified file exists.

Category:	External File
Returned data type:	Character
Note:	The returned value is true if the file exists.

Syntax

FILEEXISTS(<filename>)

Required Argument

filename
the name of the file that you are checking to find out if it exists.

Example

```
string filename
```

```
boolean rc_ok

filename="C:\doesexist.txt"

rc_ok = fileexists(filename) // outputs "true" if file exists
```

MKDIR Function

Creates a directory.

Category:	String
Returned data type:	Boolean

Syntax

MKDIR(*string*<, *create-intermediary-directories*>)

Required Argument

string
specifies the text string that contains the directory to create.

Optional Argument

create-intermediary-directories
specifies whether to create intermediary directories if they do not exist, using these values:

TRUE	specifies to create the intermediary directories.
FALSE	specifies not to create intermediary directories.

Examples

Example 1

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data directory
d mkdir(dir)
```

Example 2

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare Boolean variables for the MKDIR function call and the optional condition
```

```
boolean b
boolean d

// Set the condition "true" to create an intermediary directory if it does not exist
b=true

// Use the MKDIR function to create the new directory, including the intermediary
// directory DatQuality, if it does not exist.
d mkdir(dir,b)
```

MOVEFILE Function

Moves or renames a specified file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

MOVEFILE(<old_file, new_file_name>)

Required Arguments

old_file_name
a string representing the name of the file to be moved; this can be specified as a fixed string, field name, or expression.

new_file_name
a string representing the name (including location) where the file is moved; this can be specified as a fixed string, field name, or expression.

Details

The MOVEFILE function moves a file. The directory structure must already be in place for the function to move the file to its new location. If the target file already exists, the file is not moved and false is returned.

Example

```
string old_file_name
string new_file_name
boolean rc_ok

old_file_name = "C:\mydata_copy.txt"
new_file_name = "C:\TEMP\mydata_copy.txt"

rc_ok = movefile(old_file_name, new_file_name)
```

OPEN Function

Opens a specified file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**OPEN**(<filename, openmode>)

Required Arguments

filename
a string representing the name of the file to be opened. If the file does not exist, it is created. This parameter can be specified as a fixed string, field name, or expression.

openmode
[optional] a string representing the OPENMODE to be used. This can be specified as a fixed string, field name, or expression [a = append, r = read, w = write, rw = read and write].

Details

The open method opens the file that is provided in the filename parameter. If the file does not exist and an OPENMODE is specified containing either an "a" or "w", then the file is created. If the OPENMODE is not specified, a value of false is returned.

When WRITEBYTES and WRITELINE methods write at the end of the file, unless SEEKBEGIN, SEEKCURRENT, or SEEKEND methods are used to adjust the position in the file, the information is written at the current position in the file.

If an OPENMODE of "w" is used, the WRITEBYTES and WRITELINE methods write at the current position in the file and potentially overwrite existing information in the file.

Example

```
file myfile
if ( myfile.open("data.txt") ) then ...
```

POSITION Function

Returns the current position of the cursor in a file, which is the number of bytes from the beginning of the file.

Category:	External File
Returned data type:	Integer

Note: The returned value is an integer representing the current position (offset) from the beginning of the file.

Syntax

<fileobject>.**POSITION**(< >)

Details

The position method returns the current position of the cursor in a file. Combined with the SEEKEND() method, it can be used to determine the size of a file.

Example

```
file f
integer byte_size

f.open("C:\filename.txt", "r")
f.seekend(0) // position cursor at end of file

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.seekend(0) // position cursor at end of file

// The integer variable byte_size will have
// the size of the file in bytes
byte_size = f.position()

f.close()
```

READBYTES Function

Reads a certain number of bytes from a file.

Category: External File

Returned data type: Integer

Note: The returned value is the number of bytes actually read, or 0 on failure.

Syntax

<fileobject>.**READBYTES**(<number_of_bytes, buffer>)

Required Arguments

number_of_bytes
an integer specifying the number of bytes that need to be read from the file. This parameter can be specified as a number, field name, or expression.

buffer
a string that contains the bytes that are read. This parameter can be specified as a fixed string, field name, or expression.

Details

The READBYTES method reads the specified number of bytes from a file starting at the current position of the file pointer. The file pointer will be positioned after the last byte read. If the buffer is too small, only the first bytes from the file are put into the buffer.

This method is normally used to read binary files. The various format functions can be used to convert the binary information that was read.

Note that this method also reads EOL characters. When reading a windows text file like this:

C:\filename.txt
abc
def

A READBYTES(7, buffer) statement causes the field buffer to contain the following value "abc\n de". The value consists of all the information from the first line (3 bytes), followed by a CR character and an LF character (2 bytes on Windows, 1 byte on UNIX) that is represented by "\n". They are followed by the first 2 bytes from the second line. To read text files, use the READLINE() method.

Example

```
string input
file f

f.open("C:\filename.txt", "r")
f.readbytes(7, input)

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.readbytes(7, input)
```

READLINE Function

Reads the next line from an open file.

Category:	External File
Returned data type:	Character
Note:	The returned value is a string containing the line that was read from the file.

Syntax

fileobject.**READLINE**(< >)

Details

The READLINE method reads the next line of data from an open file. A maximum of 1024 bytes are read. The text is returned. Null is returned if there was a condition such as end of file.

Example

```
file f
string input

f.open("C:\filename.txt", "r")
input=f.readline()
f.close()
```

RMDIR Function

Deletes a directory if it is empty.

Category:	String
Returned data type:	Boolean

Syntax

RMDIR(*directory*)

Required Argument

directory
specifies the directory to remove if it is empty.

Example

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data directory
d rmdir(dir)
```

SEEKBEGIN Function

Sets the file pointer to a position starting at the beginning of the file. Returns true on success, false otherwise. The parameter specifies the position.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKBEGIN**(<position>)

Required Argument

position
an integer specifying the number of bytes that need to be moved forward from the beginning of the file. Specifying a 0 means the start of the file. This parameter can be specified as a number, field name, or expression.

Details

The SEEKBEGIN method moves the file pointer to the specified location in the file, where 0 indicates the start of the file. It returns true on success. Otherwise, false is returned. Specifying 0 means that reading starts after the first position in the file.

Example

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// return the pointer to the beginning of the file
// and read the first line again
f.seekbegin(0)
f.readline()

f.close()
```

SEEKCURRENT Function

Sets the file pointer to a position in the file relative to the current position in the file.

Category:	External File
Returned data type:	Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKCURRENT**(<position>)

Required Argument

position
an integer specifying the number of bytes that need to be moved from the current position in the file. Positive values specify the number of bytes to move forward, negative values specify the number of bytes to move backward. This parameter can be specified as a number, field name, or expression.

Details

The SEEKCURRENT method moves the file pointer from the current position in the file. This method is useful when reading binary files that contain offsets to indicate where related information can be found in the file.

Example

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// The file contains 3 bytes per record followed by a CR and LF character
// So move the pointer 3+2=5 positions back to read the beginning of
// the first line and read it again.
f.seekcurrent(-5)
f.readline()

f.close()
```

SEEKEND Function

Sets the file pointer to a position in the file counted from the end of the file.

Category: External File

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**SEEKEND**(<position>)

Required Argument

position
: an integer specifying the number of bytes that need to be back from the end of the file. Specifying a 0 means the end of the file. This parameter can be specified as a number, field name, or expression.

Details

The SEEKEND method moves the file pointer backward the number of bytes that were specified, where 0 indicates the end of the file. It returns true on success otherwise false is returned.

Example

```
file f
f.open("C:\filename.txt", "rw")

// write information to the end of the file
f.seekend(0)
f.writeline("This is the end ")
f.close()
```

WRITEBYTES Function

Writes a certain number of bytes to a file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is an integer representing the number of bytes written.

Syntax

fileobject.**WRITEBYTES**(<number_of_bytes, buffer>)

Required Arguments

number_of_bytes
: an integer specifying the number of bytes that is written to the file. This parameter can be specified as a number, field name, or expression.

buffer
: a string that contains the bytes that need to be written. This parameter can be specified as a fixed string, field name, or expression.

Details

The WRITEBYTES method writes the specified number of bytes to a file starting at the current position in the file. This method overwrites data that exists at the current position in the file. If the current position in the file plus the number of bytes to be written is larger than the current file size, then the file size is increased.

If buffer is larger than number_of_bytes specified, then only the first number_of_bytes from buffer is written. The file needs to be opened in Write or Append mode for this method to work. The method returns the actual number of bytes written.

This method is normally used to write binary files. To write text files, the WRITELINE() method can be used.

Example

```
string input
file f

string = "this is longer than it needs to be"
f.open("C:\filename.txt", "rw")
// This will write to the beginning of the file
// Only the first 10 bytes from the string will be written
// If the file was smaller than 10 bytes it will be automatically
// appended
f.writebytes(10, input)

f.close()
```

WRITELINE Function

Writes a line to a file.

Category:	External File
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

fileobject.**WRITELINE**(<string>)

Required Argument

string
a string specifying the information that needs to be written to the file. This parameter can be specified as a fixed string, field name, or expression.

Details

The WRITELINE() method writes the string at the current position in the file. This method overwrites data that exists at the current position in the file. If the current position in the file plus the length of the string is larger than the current file size, then the file size is increased.

The file needs to be opened in Write or Append mode for this method to work.

Example

```
file f
```

```
f.open("C:\filename.txt", "a")
f.writeline("This text will be appended to the file")

f.seekbegin(0)
f.writeline("Using seekbegin(0) and Append will
still cause the info to be written at the start of the file")

f.close()
```

Chapter 9

Incoming Data Functions

Overview

The EEL provides built-in functions that enable you to evaluate data coming in from a parent node, as well as set the value of a field, and determine the type of field and the maximum length.

The FIELDCOUNT, FIELDNAME, and FIELDVALUE functions enable you to dynamically access values from the parent node without knowing the names of the incoming fields.

Dictionary

FIELDCOUNT Function

Accesses values dynamically from the parent node without knowing the names of the incoming fields. Returns the number of incoming fields.

Category:	Data Input
Returned data type:	Integer
Note:	The returned value is an integer, representing the number of incoming fields from the parent node.

Syntax

FIELDCOUNT(< >)

Details

Provides a way of dynamically accessing values from the parent node without knowing the names of the incoming fields. The function returns the number of incoming fields.

Example

```
// Declare a hidden integer for the for loop, initializing it to 0
hidden integer i
i = 0

// Increment through the data once for each column of data in the input data
for i = 1 to fieldcount()
```

FIELDNAME Function

Returns the name of a specific field output from the parent node.

Category:	Data Input
Returned data type:	Character

Syntax

FIELDNAME(*index*)

Required Argument

index
is the index into the incoming fields

Examples

Example 1

```
// Declare a string variable to contain the field name
String Field3

// Use the Fieldname function to get the third field in the incoming data
Field3 = Fieldname(3)
```

Example 2

```
// Declare a hidden integer for the for loop, initializing it to 0
hidden integer i
i = 0
// Declare a string variable to contain the column names
string column_name
```

```
// Create a table with a row for each column in the input data source
for i = 1 to fieldcount()
begin
   column_name = fieldname(i)
pushrow()
end
```

FIELDTYPE Function

Returns the field type of a field output from the parent node. If the second parameter is supplied, it is set to the length in chars if the field type is string.

Category:	Data Input
Returned data type:	Character

Syntax

FIELDTYPE(<(index, length>)

Required Argument

<index>
specifies the index into the incoming fields from the parent node. The second parameter is optional and set to the maximum string length in characters if the field type is a string.

Optional Argument

<length>
specifies an integer that contains the length of the field if the field is of type string.

Details

The FIELDTYPE function determines the type and, the maximum length in characters (for string fields) based on its index in the list of fields coming from the parent node. It returns a string representation of the field type (for example, integer or date).

Example

```
// Declare a hidden integer for the for loop, initializing it to 0
hidden integer i
i = 0

// Increment through the data a number of times equal to the number
// of fields in the data
for i = 1 to fieldcount()

//Check the type of each field in the data and take some action
if fieldtype(i) == 'Date' then
```

FIELDVALUE Function

Returns the value of a specified field as a string.

Category: Data Input

Syntax

*string***FIELDVALUE**(<integer>)

Required Argument

<integer>
specifies the index into the incoming fields.

Examples

Example 1

```
// Declare a string variable to contain the third field value
String Value_Field3

// Use the fieldvalue function to get the
// value in the third field of the
// incoming data
Value_Field=fieldvalue(3)
```

Example 2

```
// Declare a hidden integer for the for loop, initializing it to 0
hidden integer i
i = 0

// Checks each field to see if the field is a name field and the value is numeric
for i = 1 to fieldcount()
begin
   if instr(lower(fieldname(i)),'name') then
   if isnumber(fieldvalue(i)) then
   return true
end
```

READROW Function

Reads the next row of data from the step above and fills the variables that represent the incoming step's data with the new values. It returns false if there are no more rows to read.

Category: Data Input

Returned data type: Integer

Note: The returned value is a Boolean value representing whether there are values in the incoming step's data, true = there are still rows in parent node; false = no more rows to read.

Syntax

READROW(< >)

Details

The READROW function is a Data Job-only function.

Note: The READROW function has no effect when called from a pre- or post-expression. When called from a pre-group or post-group expression, it can cause undesirable results.

Example

Assume that this step is below a step with a name field and the step outputs four rows, "John", "Paul", "George", and "Igor":

```
// Declare a string to contain the "old" or "previous" value
string oldname

// Set the value of OLDNAME to whatever is in NAME
oldname=name
     // Name has the value "John', oldname also contains "John"

// Use the READROW function to read in the next record
readrow()
     // OLDNAME is still "John", but NAME is now "Paul"
```

ROWESTIMATE Function

Sets the estimated total number of rows to be reported by this step.

Category: Data Input

Returned data type: Integer

Note: The returned value is a Boolean value where 1= success and 0 = error.

Syntax

ROWESTIMATE(<integer>)

Optional Argument

<integer>
specifies an integer that contains an estimate for the total numbers of rows that are output from the step.

Details

The ROWESTIMATE function is used by data jobs to estimate the number of records that will be output from the step.

Example

```
// Declare a hidden integer for the number of output rows
hidden integer nrows

// Set the number of rows for the function
nrows=100

// This function estimates and sets the # of records that this step will report
rowestimate(nrows)
```

SETEOF Function

Sets status to end of file (EOF), preventing further rows from being read in the step. If the parameter is true, the pushed rows are still returned.

Category:	Data Input
Returned data type:	Integer
Note:	The returned value is a Boolean value where 1= success and 0 = error.

Syntax

SETEOF(<*return_pushrow*>)

Optional Argument

return_pushrow
Boolean value; to specify whether pushed rows are still returned

Details

When SETEOF() is called, the node does not read any more rows from the parent node. If generate rows when no parent is specified is checked, the node stops generating rows. Furthermore, if any rows have been pushed using PUSHROW(), they are discarded, and further calls to PUSHROW() have no effect. The exception to this is if SETEOF(true) is called. In this case, any pushed rows (whether pushed before or after the call to SETEOF() are still returned to the node below. Notably, if further PUSHROW() calls occur after SETEOF(true) is called, these rows are returned as well. Also note that after SETEOF() is called, the post-group expression and the post expression are still executed.

Example

```
seteof()
```

SETFIELDVALUE Function

Sets the value of a field based on its index in the list of fields coming from the parent node. This is useful for enumerating through the fields and setting values.

Category: Data Input

Syntax

SETFIELDVALUE(<integer,any>)

Required Arguments

integer
index into the incoming fields.

any
value that you want to set the field to.

Details

The SETFIELDVALUE function sets the value of a field based on its index in the list of fields coming from the parent node. This is useful for enumerating through the fields and setting values.

Example

```
// Declare a hidden integer for the for loop, initializing it to 0, and a hidden date fi
hidden integer i
i = 0
hidden date Date_Field

// Checks each field to see if it is a date field
for i = 1 to FieldCount()
if FieldType(i) == 'Date' then

begin
   Date_Field= FieldValue(i)
// If the date is in the future, then use SETFIELDVALUE to set the value to null
   if Date_Field > today()
   SetFieldValue(i,null)
end
```

Chapter 10

Information/Conversion Functions

Overview

The following information and conversion functions are available for the Expression Engine Language (EEL).

Dictionary

DETERMINE_TYPE Function

Returns the type of data that the input string represents.

Category:	Date
Returned data type:	String

Syntax

DETERMINE_TYPE

Required Arguments

string
is the string of data.

returns
the data type the input string represents.

Details

This function analyzes a string to determine whether it is one of the following options: string, integer, Boolean, date, or real.

Examples

Example 1

```
1000
Results: integer
```

Example 2

```
1000.5
Results: real
```

GEODISTANCE_COSINE Function

Computes the distance between two geographical points. This function provides results faster and provides an acceptable level of accuracy for most cases.

Category: Information and Conversion

Syntax

GEODISTANCE_COSINE(<lat1, lon1, lat2, lon2>)

Required Arguments

lat1
latitude for first location, in decimal degrees. South latitudes are negative.

lon1
longitude for first location, in decimal degrees. East longitudes are positive.

lat2
latitude for second location, in decimal degrees. South latitudes are negative.

lon2
longitude for second location, in decimal degrees. East longitudes are positive.

Details

The GEODISTANCE_COSINE function is used to compute the distance between two geographical points.

Note: Each point is represented using latitude and longitude in decimal degrees. South latitudes are negative and east longitudes are positive. The function returns -1 when there is an error.

Example

```
geodistance_cosine(21.349017, -45.566906, -35.203839, 19.996602)
```

GEODISTANCE_HAVERSINE Function

Computes the distance between two geographical points. This formula provides greater accuracy, particularly for shorter distances.

Category: Information and Conversion

Note: This function provides better accuracy than the GEODISTANCE_COSINE function, but is slower.

Syntax

GEODISTANCE_HAVERSINE(<lat1, lon1, lat2, lon2>)

Required Arguments

lat1
latitude for first location, in decimal degrees. South latitudes are negative.

lon1
longitude for first location, in decimal degrees. East longitudes are positive.

lat2
latitude for second location, in decimal degrees. South latitudes are negative.

lon2
longitude for second location, in decimal degrees. East longitudes are positive.

Details

The GEODISTANCE_HAVERSINE function is used to compute the distance between two geographical points.

Note: Each point is represented using latitude and longitude in decimal degrees. South latitudes are negative and east longitudes are positive. The function returns -1 when there is an error.

Example

```
geodistance_haversine(21.349017, -45.566906, -35.203839, 19.996602)
```

ISALPHA Function

Returns a true value if the expression is a string made up entirely of alphabetic characters.

Category: Information and Conversion

Returned data type: Integer

Note: The returned value is a Boolean value, true if the "in_string" contains only alpha characters. Otherwise, the value is false.

Syntax

ISALPHA(<in_string>)

Required Argument

in string
a string of characters that is searched for any alphabetic characters.

Details

The ISALPHA function returns true if "in_string" is determined to be a string containing only alpha characters.

Examples

Example 1

```
// Expression
string letters
letters="lmnop"
string mixed
mixed="1a2b3c"

string alphatype
alphatype=isalpha(letters) // returns true
string mixedtype
mixedtype=isalpha(mixed) // returns false
```

Example 2

```
string all_Alpha
all_Alpha="abcdefghijklmnoprstuvyz"

string non_Alpha
non_Alpha="%&)#@*0123456789"

string error_message1
string error_message2

if (NOT isalpha(all_Alpha))
   error_message1 ="all_Alpha string contains alpha numeric characters"
else
   error_message1 ="all_Alpha string contains alpha numeric characters"

if(isalpha(non_Alpha))
   error_message2= "non_Alpha string contains alpha numeric characters"
else
   error_message2= "non_Alpha string does not contain alpha numeric characters"
```

Example 3

```
string all_Alpha
string error_message
all_Alpha="abcdefghijklmnopqrstuvwxyz"
if (isalpha(all_Alpha))
   begin
        error_message= "alpha strings were identified as alpha"
   end
```

ISBLANK Function

Checks if an argument contains a blank, empty value. When the argument value is blank, the function returns true. Otherwise, it returns false.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value.

Syntax

<boolean>**ISBLANK**(<argvalue>)

Required Argument

argvalue
is a string.

Details

The ISBLANK function takes the following argument types: string.

Example

```
string x
string y
date z
string error_message1
string error_message2

x="Hello"

if(isblank(x) )
   error_message1 = "x is blank"
else
   error_message1= "x is not blank"

if( isblank(y) )
   error_message2 =" String y value is blank"
else
   error_message2 =" String y value is not blank"
```

ISNULL Function

Checks if an argument value contains a null value. When the argument value is null, the function returns true. Otherwise, it returns false.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value.

Syntax

<boolean>**ISNULL**(<argvalue>)

Required Argument

argvalue
string, date, integer, real, Boolean.

Details

The ISNULL function takes the following argument types: string, date integer, real, Boolean.

Examples

Example 1

```
// Expression
if State <> "NC" OR isnull(State)
   return true
else
   return false
```

Example 2

```
integer x
string y
string error_message1
string error_message2

y="Hello"

if(isnull(x) )
   error_message1 = "Integer x is null"
else
   error_message1= "Integer x is not null"

if( isnull(y) )
   error_message2 =" String y value is null"
else
   error_message2 =" String y value is not null"
```

ISNUMBER Function

Checks if an argument value contains a numerical value. When the argument value is a number, the function returns true. Otherwise, it returns false.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value.

Syntax

argvaluebooleanISNUMBER()

Required Argument

argvalue
string

Details

The ISNUMBER function takes the following argument types: string.

Example

```
string x
string y
string z
string error_message1
string error_message2
string error_message3

x ="5"
y="Hello"
z="01/01/10"

if(isnumber(x) )
   error_message1 = "String x is a number"
else
   error_message1= "String x is not a number"

if( isnumber(y) )
   error_message2 =" String y value is a number"
else
   error_message2 =" String y value is not a number"

if( isnumber(z) )
   error_message3 = "String z value is a number"
else
   error_message3 = "String z value is not a number"
```

IS_PREVIEW_MODE Function

Returns TRUE if the job is executed in preview mode. Otherwise, it returns FALSE.

Category:	Repository
Returned data type:	Boolean

Syntax

IS_PREVIEW_MODE( )

Without Arguments

There are no arguments for this function.

LOCALE Function

Sets the locale.

Category:	Information and Conversion
Returned data type:	String
Notes:	The returned value is a string of the current locale setting. This function affects certain operations such as converting and date operations. This function returns the previous locale. If no parameter is passed, the current locale is returned. The locale setting is a global setting.

Syntax

<string>**LOCALE**(< >)

<string>**LOCALE**(<“locale_string”>)

Details

If a parameter is specified, it is set. Otherwise, it is retrieved. If setting, the old locale is retrieved.

The following values can be set in the LOCALE() function:

- You can use a two-character abbreviation for the US and UK locales
- A three-character abbreviation can be used for some countries, like GER, FRA, or DEU

Here are some examples:

```
my_locale = locale("DEU")
my_locale = locale("German")
my_locale = locale("German_Germany")
my_locale = locale("German_Germany.1252")
```

The LOCALE() function returns the current setting using Country_Language.codepage notations.

Note: If you use ("#iso8601"), the date is set in the ISO8601 format.

Example

```
string currentSetting
string newSetting

currentSetting = locale();

newSetting = locale("FRA");
```

TOBOOLEAN

Converts the argument to a Boolean value.

Category:	Information and Conversion
Returned data type:	Integer
Note:	The returned value is a Boolean value that is returned if the argument value can be converted to a Boolean value

Syntax

boolean**TOBOOLEAN**(value<>)

Required Argument

value
is passed in as one of the following: real, integer, string, or date.

Example

```
boolean convertedValue
integer result
result = 1
convertedValue = toboolean(result)
Print (convertedValue)
```

TYPEOF Function

Identifies the data type of the passed in value.

Category:	Information and Conversion
Returned data type:	Character
Note:	The returned value is one of the following strings: Boolean variable return Boolean, integer variable return integer, real variable return real, string variable return string.

Syntax

<string> **TYPEOF**(in_value)

Optional Argument

in value
variable that is evaluated.

Examples

Example 1

```
// Expression
string hello
hello="hello"

boolean error
error=false

// variable that will contain the type
string type
type=typeof(hello)

// type should be string
if(type<>"string") then
   error=true
```

Example 2

```
string content
content = "Today is sunny"

hidden integer one
one =1

hidden real pi
pi=3.1415962

hidden boolean test
test=false

hidden string type

type= typeof(content);

if (type == "string")
   begin
        error_message="The data type for variable 'Content' is string"
   end

type=typeof(one)
if (type == "integer")
   begin
        error_message="The data type for variable 'one' is integer"
   end
```

```
type= typeof(pi);

if (type == "real")
   begin
        error_message="The data type for variable 'real' was real"
   end

type= typeof(test);

if (type == "boolean")
   begin
        error_message="The data type for variable 'test' was boolean"
   end
```

Chapter 11

Logging Functions

Overview

The following logging functions are available for the Expression Engine Language (EEL).

- LOGMESSAGE
- PRINT
- RAISEERROR
- SENDNODESTATUS

Dictionary

LOGMESSAGE Function

Prints a message to the log.

Category:	Logging
Returned data type:	Integer
Note:	The returned value is a Boolean value, always true for the LOGMESSAGE function.

Syntax

LOGMESSAGE(string message)

Required Argument

message
a string representing the text of the message to be written to the log

Details

The LOGMESSAGE function is used to send a message to the log file.

Example

```
logmessage("This message will go to the log file")
```

PRINT Function

Prints the string to the step log.

Category:	Logging
Returned data type:	Integer
Notes:	The returned value is a Boolean value, which is always true. If the second parameter is true, no linefeeds will be appended after the text.

Syntax

PRINT(string, no linefeed)

Required Argument

string
is the text to be printed; this can be specified as a text constant or a field name.

Optional Argument

no linefeed
the second Boolean determines whether linefeeds will be printed to the log after the text.

Details

The PRINT function writes text to the node summary.

Example

```
// Declare a string variable to contain the input value
string input

// Set the string variable to a value
// Use the PRINT function to write a note to the log
input='hello'
print('The value of input is ' & input)
```

RAISEERROR Function

Prints a message to the run log.

Category:	Logging
Returned data type:	Integer
Note:	There is no return value for this function since an error occurs when this is called and the expression stops processing.

Syntax

<boolean>**RAISEERROR**(string usererror)

Required Argument

usererror
is a string that is printed to the jobs output.

Details

The RAISEERROR function raises a user-defined error. Users can define a condition and then use RAISEERROR to stop the job and return an error message when the condition occurs. This is useful for evaluating problems unique to an installation. The user can then search for the error message to determine whether the associated condition was responsible for stopping the job.

Example

```
raiseerror("user defined error")
```

SENDNODESTATUS Function

Sends a node status.

Category:	Node
Returned data type:	Boolean

Syntax

SENDNODESTATUS(status key, statusvalue)

Required Arguments

status key
a string representing the name of the status key available value is:

- “_PCT_COMPLETE” //percent complete

- "_OVERALL" //overall status

status value
a string representing the value of the status key

returns
Boolean; true if the SENDNODESTATUS was successful; false otherwise

Details

Note: The SENDNODESTATUS function is applicable only in a process job.

The SENDNODESTATUS function tells the process job engine of the status of the node. If an expression is going to run for a long time it is a good idea for it to send its status. It takes two parameters, a status key, and a status value.

Available keys are:

```
"__PCT_COMPLETE" // percent complete
"__OVERALL" // overall status (suitable for displaying on the ui with a node)
```

Example

The following statements illustrate the SENDNODESTATUS function:

```
integer i
for i = 1 to 100
begin
   sleep (100)
   sendnodestatus ("__PCT_COMPLETE", i)
end
```

Chapter 12

Macro and Variable Functions

Overview

Macros (or variables) are used to substitute values in a job. This might be useful if you want to run a job in different modes at different times. For example, you might want to run the job every week, but read from a different file every time it runs. In this situation, you would specify the filename with a macro rather than the actual name of the file. Then, set that macro value either on the command line (if running in batch) or by using another method.

About Macros and Variables

Multiple macro files are supported along with the concept of user- and system-level macros, in a specific order.

System macros are defined in the dfexec_home location and displayed through the user interface, but they cannot be added or edited. User settings are stored in the %apdata% location. You can view, add, or edit through the user interface. Changes to the system-level macro cause an override where the new value is written to the user location. To promote this change, you must update the system location outside of Data Management Studio. New system macros and macro files must be created outside the software.

Load order is important because technical support can use load order to force a macro to be a specific value. In the following, the base directory is defined by dfexec_home. In a typical installation, this is the root directory where DataFlux Data Management Studio is installed.

Command line declarations override environment variables which in turn override macro variable values declared in any of the system or user configuration files. Refer to the DataFlux Data Management Studio Online Help for more information about using macro variables. The results from the Expression node are determined by the code in the Expression Properties dialog box.

Using Macros and Variables

All of the settings in the DataFlux configuration file are represented by macros in the Data Job Editor. For example, the path to the QKB is represented by the macro DQ/QKB. To use a macro in a job, enter it with double percent signs before and after the value. For example:

Example Code 12.1 *Old Value:*

```
C:\myfiles\inputfile01.txt
```

Example Code 12.2 *Using Macros, You Enter:*

```
%%MYFILE%%
```

You can also use the macro to substitute some part of the parameter as in this example, C:\myfiles\%%MYFILE%%. A macro can be used anywhere in a job where text can be entered. If a data job step (such as a drop-down list) prevents you from entering a macro, go to the **Advanced** tab and enter it there. After you enter a macro under the **Advanced** tab, you get a warning if you try to return to the standard property dialog box. Depending on your macro, you might need to avoid the standard property dialog box and use the advanced dialog box thereafter. If the property value is plain text, you can return to the standard dialog box.

You can choose to use variables for data input paths and filenames in the data job Expression node. You can declare macro variables by any of these ways:

- entering them in the Macros folder of the Administration riser bar
- editing the macro.cfg file directly
- specifying a file location when you launch a job from the command line

When you add macros using the Administration riser, Data Management Studio directly edits the macro.cfg file. If you edit the macro.cfg file directly, you can also add multiple comments.

Command line declarations override the macro variable values declared in app.cfg. The results from Expression are determined by the code in the Expression Properties dialog box.

Specifically, the value of a macro is determined in one of the following ways:

- In the first case, if you are running in batch in Windows, the -VAR or -VARFILE option lets you specify the values of the macros. For example:

  ```
  -o key1=value1 -o key2=value2
  -C "C:\mymacros.txt"
  ```

  Note: The return code can be checked by creating a batch file, and checking the error level in the batch file by using the following:

  ```
  IF ERRORLEVEL [return code variable] GOTO
  ```

  If the return code is not set, it returns a 0 on success and 1 on failure. The return code can be set using the RETURN_CODE macro.

- In the second case, the file contains each macro on its own line, followed by an equal sign and the value.
  - If running in batch on UNIX, all current environment variables are read in as macros.

- If running data jobs in Windows, the values specified in **Tools** > **Options** > **Global** are used.
- If running the Data Management Server, the values can be passed in the SOAP request packet.
- If using an embedded job, the containing job can specify the values as parameters.
- The app.cfg file can be used to store additional values. These values are always read regardless of which mode is used.

Using GETVAR() and SETVAR()

Macro variable values can be read within a single function using the %my_macro% syntax. If you are using more than one expression in your job, use GETVAR() to read variables and SETVAR() to read and modify variables. With GETVAR() and SETVAR(), changes to the value persist from one function to the next. Note that changes affect only that session of the Data Job Editor and are not written back to the configuration file.

The following table contains information about predefined macros:

Predefined Macro	Description
_JOBFILENAME	The name of the current job.
_JOBPATH	The path of the current job.
_JOBPATHFILENAME	The path and filename to the current job.
TEMP	The path to the temporary directory.

Note: For SETVAR(), if you set a macro in an expression step on a page, the new value cannot be reflected in nodes on that page. This condition is because those nodes have already read the old value of the macro and might have acted upon it (such as opening a file before the macro value was changed). This issue arises only from using SETVAR(). Thus, SETVAR() is useful only for setting values that are read on following pages or from other expression nodes with GETVAR().

Dictionary

GETVAR Function

Returns run-time variables. These are variables that are passed into DataFlux Data Management Studio on the command line using -c or -o.

Category:	Macro Variable
Returned data type:	String

Syntax

GETVAR(*string1*<, *string2*>)

Required Argument

string1
specifies the name of the variable to return. *string1* is not case sensitive.

Optional Argument

string2
specifies the value that is returned if the variable does not exist.

Example

```
testInParam=getvar("DF_String_Input")
File f
f.open("%%DataFlowTargets%%DF_String_Input.txt", "w")
f.writeline("DF_String_Input = "&testInParam)
seteof()
f.close()
```

SETVAR Function

Sets the Data Job macro variable value, indicated by the first parameter, to the values in the second parameter.

Category:	Macro Variable
Returned data type:	Boolean

Syntax

SETVAR(*macroname*, *value*)

Required Arguments

macroname
specifies the name of the Data Job macro variable to set; this can be specified as a string.

value
specifies the value to set to *macroname*; this value can have a data type of real, integer, string, date, or Boolean.

Details

This function return TRUE if it completed successfully. Otherwise, it returns FALSE.

Example

```
string macroName
```

```
macroName = "myMacro"
newValue = 10

success=setvar(macroName, newValue)
```

VAREVAL Function

Returns the value of a variable.

Category:	String
Returned data type:	String
Tip:	The VAREVAL( ) function must look up field names each time it is called. Use the VAREVAL( ) function sparingly to avoid performance degradation.

Syntax

VAREVAL (*string*)

Required Argument

string
specifies a string that resolves to the name of a variable.

Details

You can use the VAREVAL function to help you dynamically select the value of different fields. First, write the code that creates the names of the fields that you want to access. Then, specify the field name in the VAREVAL function to access the value of that field. See the following example.

Example

Assume you have the following fields in your data: field_1, field_2, field_3, field_4 and field_5.

```
// Declare the string values for the function
string field_number
string field_value
// Declare a hidden integer as a counter
hidden integer n
// Loop trough all 5 variables in an input data source
for n=1 to 5
// Output the value in each of the fields field_1 through field_5
begin
     field_number='field_' & n
     field_value=vareval(field_number)
     pushrow()
end
// Return false to prevent the last row from showing up twice
return false
```

VARSET Function

Sets a variable to a value. This allows the name of the variable to be immediately evaluated at run time.

Category:	Macro Variable
Returned data type:	Boolean

Syntax

VARSET(*variablename*, *value*)

Required Arguments

variablename
specifies the name of a variable that is to receive the value. *variablename* can be specified as a string.

Note: The string value of this parameter is used to figure out which variable to alter.

value
specifies the value to set to *macroname*; *value* can have a data type of real, integer, string, date, or Boolean.

Details

The function return TRUE if it successfully completes. Otherwise, it is set to FALSE.

Example

```
string myvar1
string myvar2
string myvar3
integer x
for x = 1 to 3
  varset("myvar" & x, "value is " & x)

// This should result in each string variable receiving
// a different value as assigned.
```

Chapter 13

Mathematical Functions

Overview

The following mathematical functions are available for the Expression Engine Language (EEL).

Dictionary

ABS Function

Returns the absolute value of a number.

Category: Mathematical

Returned data type: Real

Note: The ABS function returns a nonnegative number that is equal in magnitude to the magnitude of the argument.

Syntax

ABS(*argument*)

Required Argument

argument
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Examples

Example 1

Statements	Results
x = abs(3.5)	// outputs 3.5
x = abs(-7)	// outputs 7
x = abs(-3*1.5)	// outputs 4.5

Example 2

```
real seconds_diff
seconds_diff = abs((date1 - date2) * 86400)
// The number 86400 represents the total number of seconds in a day
```

CEIL Function

Returns the smallest integer that is greater than or equal to the argument.

Category: Mathematical

Returned data type: Real

Syntax

CEIL(*argument*)

Required Argument

argument
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Details

This is also called rounding up (ceiling).

Example

```
x = ceil(3.5)
// outputs 4

x = ceil(-3.5)
```

```
// outputs -3

x = ceil(-3)
// outputs -3

x = ceil(-3*1.5)
// outputs -4
```

FLOOR Function

Returns the largest integer that is less than or equal to the argument.

Category: Mathematical

Returned data type: Real

Syntax

FLOOR(*argument*)

Required Argument

argument
specifies a value that has a data type of real; this can be specified as a numeric constant, field name, or expression.

Details

This is also called rounding down.

Example

```
x = floor(3.5)
// outputs 3

x = floor(-3.5)
// outputs -4

x = floor(-3)
// outputs -3

x = floor(-3*1.5)
// outputs -5
```

MAX Function

Returns the maximum value of a series of values.

Category: Mathematical

Returned data type: Real

Syntax

MAX(*argument1*< , *argument2*, …>)

Required Argument

argument1
specifies a value that has a real data type; this can be specified as a numeric constant, field name, or expression.

Optional Argument

***argument2*, …**
specifies one or more values that have a real data type; these can be specified as a numeric constant, field name, or expression.

Details

The function returns NULL if all values are NULL.

Example

```
x = max(1, 3, -2)
 // outputs 3

x = max(1, null, 3)
 // outputs 3

x = max(-3)
// outputs -3

x = max(4, -3*1.5)
 // outputs 4
```

MIN Function

Returns the minimum value of a series of values.

Category:	Mathematical
Returned data type:	Real

Syntax

MIN(*argument1*<, *argument2*, …>)

Required Argument

argument1
specifies a value that has a data type of real; this can be specified as a numeric constant, field name, or expression.

Optional Argument

***argument2*, …**
specifies one or more values that have a real data type; these can be specified as a numeric constant, field name, or expression.

Details

The function returns NULL if all values are NULL.

Example

```
x = min(1, 3, -2) // outputs -2
x = min(1, null, 3) // outputs 1
x = min(-3) // outputs -3
x = min(4, -3*1.5) // outputs -4.5
```

POW Function

Raises a number to the specified power.

Category: Mathematical

Returned data type: Real

Syntax

POW(*x*, *y*)

Required Arguments

x
specifies a value that has a data type of real and indicates the base number to raise; this can be specified as a numeric constant, field name, or expression.

y
specifies a value that has data type of real and indicates the power to raise to; this can be specified as a numeric constant, field name, or expression.

Details

The POW function raises *x* to the power *y*, x^y.

Example

Statements	Results
x = pow(5,2)	// outputs 25
x = pow(5,-2)	// outputs 0.04
x = pow(16,0.5)	// outputs 4

ROUND Function

Rounds a number to the nearest number with the specified decimal places.

Category:	Mathematical
Returned data type:	Real

Syntax

ROUND(*argument*< , *decimals*>)

Required Argument

argument

a value with a data type of real that can be specified as a numeric constant, field name, or expression.

Optional Argument

decimals

specifies a numeric constant, field name, or expression that indicates the number of decimal places to provide in the result of the rounding operation. A positive value for decimals is used to round to the right of the decimal point. A negative value is used to the left of the decimal point.

Default	0

Example

```
x = round(1.2345,1)
 // outputs 1.2

x = round(1.449,2)
 // outputs 1.45

x = round(9.8765,1)
 // outputs 9.9

x = round(9.8765)
 // outputs 10
```

Chapter 14 Node Functions

Overview

The following node functions are available for the Expression Engine Language (EEL).

Dictionary

PCTCOMPLETE Function

Reports to the user interface the estimated percent complete.

Category:	Node
Returned data type:	Integer

Syntax

PCTCOMPLETE( )

Without Arguments

There are no arguments for this function.

Details

This function is used to view the estimated percent complete for process flow or data flow. The function returns an integer that indicates the estimated percentage complete.

RAISEEVENT Function

Raises the specified event; pass an arbitrary number of key-value pairs for event data. The event is raised to the process flow level where it can be selected by an event node at that level.

Category:	Node
Returned data type:	Boolean

Syntax

RAISEEVENT(*event_name* <, *key1*, *value1*, *key2*, *value2*, …>)

Required Argument

event_name
specifies a string that indicates the name of the event to be raised.

Optional Argument

key1, value1 , key2, value2, …
specifies one or more key-value pairs that indicates event data.

Details

The RAISEEVENT function raises an event. The first parameter is the name of the event to raise. There is another process job node to catch the event. In that node, you can specify the name of the event to catch. Subsequent parameters are event data in the form of key-value. The keys and values are arbitrary, but they must come in pairs. For example, you might have the function with three parameters (name, key, value) or five parameters (name, key, value, key, value) and so on.

If the function completes successfully, it returns TRUE. Otherwise, it returns FALSE.

Example

```
//!event INFO_MISSING
raiseevent("INFO_MISSING", "FILE", "OK", "FIELDS", "NO DATE")
```

SENDNODESTATUS Function

Sends a node status.

Valid in:	Only in a process job.
Category:	Node
Returned data type:	Boolean

Syntax

SENDNODESTATUS("*status-key*", *status-value*)

Required Arguments

"status-key"
specifies one of the following strings to indicates the name of the status that is being reported:

_PCT_COMPLETE	the percentage complete
OVERALL	the overall status, which appears in the application interface as node status

status-value
specifies a string that indicates the value of *status-key*.

Details

The SENDNODESTATUS function tells the process job engine of the status of the node. If an expression is going to run for a long time it is a good idea for it to send its status. The function returns TRUE if the function completes successfully. Otherwise, it returns FALSE.

Example

```
integer i
for i = 1 to 100
begin
   sleep (100)
   sendnodestatus ("__PCT_COMPLETE", i)
end
```

SETOUTPUTSLOT Function

Sets output slot to slot. This becomes the output slot when the expression exits.

Valid in:	Only in a process job.
Category:	Node
Returned data type:	Boolean

Syntax

SETOUTPUTSLOT(*slot*)

Required Argument

slot
specifies an integer that indicates the active output slot when the expression node exits.

Details

The SETOUTPUTSLOT function tells the node (the expression node in which it is running) to exit on the specified slot. In a process job, if you follow a node by two other nodes, you specify a slot for each, for example, 0 and 1. If you run SETOUTPUTSLOT(1), it tells the process job to continue with the node that is linked at slot 1. If SETOUTPUTSLOT is not called, it exits on 0 by default.

The function returns TRUE if it completed successfully. Otherwise, it returns FALSE.

Example

```
if tointeger(counter)<= 5 then setoutputslot(0)
                          else setoutputslot(1)
```

UNIQUEID Function

Returns a unique identifier.

Valid in:	Process flows and data flows
Category:	Node
Returned data type:	String

Syntax

UNIQUEID( )

Details

This UNIQUEID function string is used to uniquely identify a row of data on a single machine or multiple machines.

Chapter 15

Regular Expression Functions

Overview

The regular expression (regex) object enables you to perform regular expression searches of strings in Expression Engine Language (EEL).

Dictionary

COMPILE Function

Compiles a valid regular expression using the specified encoding.

Category: Regular Expression

Returned data type: Integer

Syntax

r.**COMPILE**(*regex* <,*encoding*>)

Required Argument

regex
specifies a Perl-compatible regular expression.

Optional Argument

encoding
specifies a string that defines the encoding constant shown Encoding. Use a value from the Encoding column.

Default	The default encoding for the operating system.

Details

The compile function is used with a regular expression object. You must define a regular expression object first as a variable before you can use COMPILE( ) to compile a PERL-compatible regular expression. Regular expressions can be used to do advanced pattern matching (and in some cases pattern replacement). Use the other functions listed below to find patterns in a given string (which can be a variable), to determine the length of matching patterns, and to replace patterns.

The function returns 1 if the regular expression compilation is successful. Otherwise, it returns 0. Failure could be due to an incorrectly formatted regular expression or possibly an invalid encoding constant.

For performance reasons, it is best to compile a regular expression in a preprocessing step in the **Expression** node. This means that the regular expression is compiled just once before data rows are processed by the **Expression** node.

Note: The sample code in this section generally places the COMPILE() function on the **Expression** tab with the rest of the expression code for clarity.

In some cases, you might need to compile the regular expression before every row is evaluated. For example, you can use a variable to define the regular expression that you want to compile. The variable might come from the data row itself, and you would need to recompile the regular expression for each row to have the pattern searching work correctly.

Take care to design regular expressions that find patterns only for which you want to search. Poorly written regular expression code can require a lot of additional processing that can negatively impact performance.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//You must define a regex object
regex r
//Then compile your regular expression
//This example will match any single digit in an input string
r.compile ("[0-9]","ISO-8859-1")
// Terminate the expression node processing
seteof()
```

See Also

- "FINDFIRST Function" on page 117
- "FINDNEXT Function" on page 118
- "MATCHLENGTH Function" on page 119
- "MATCHSTART Function" on page 120
- "REPLACE Function" on page 122
- "SUBSTRINGCOUNT Function" on page 123
- "SUBSTRINGLENGTH" on page 125
- "SUBSTRINGSTART Function" on page 126

FINDFIRST Function

Searches the specified string for a pattern match using an already compiled regular expression.

Category:	Regular Expression
Returned data type:	Boolean

Syntax

r.**FINDFIRST**(*input*)

Required Argument

input

specifies the string value in which you want to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to the expression node as a column from a previous node (MyValue or "My Value").

Requirement *input* must not be NULL or blank.

Details

The FINDFIRST function indicates whether one or more pattern matches were found in the input. This function can be used to enter a logical loop that pulls out a series of matched patterns from an input string. The function returns TRUE if a pattern match is found. A FALSE return value indicates that no match is found and that processing can continue.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//You must define a regex object
regex r
```

```
//Then compile your regular expression. This one will match any single
// uppercase letter
r.compile("[A-Z]")
// If a pattern match is found this will evaluate to 1 (TRUE)
if r.findfirst("Abc")
// Print the output to the statistics file. You must run
// this job for stats to be written. A preview will not generate
// a message in the log.
print("Found match starting at " & r.matchstart() & " length " & r.matchlength())
// Terminate the expression node processing
seteof()
```

FINDNEXT Function

Continues to search the string for the next match after using the FINDNEXT() function.

Category:	Regular Expression
Returned data type:	Boolean

Syntax

r.**FINDNEXT**(*input*)

Required Argument

input

specifies the string value in which you want to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to your expression node as a column from a previous node (MyValue or "My Value").

Requirement *input* must not be NULL or blank.

Details

The FINDNEXT function indicates that another pattern match has been found after FINDFIRST() has been used. Using a "While" statement loop lets you iterate through all potential pattern matches using this function as long as the return value is equal to true.

The function returns TRUE if a pattern match was found. Otherwise, it returns FALSE.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF( ) function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some string variables
string MyString
string MySubString
```

```
// Set one to some sample input
MyString = "DwAwTxAxFyLyUzXz"

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]")
// Find the first pattern match
if r.findfirst(MyString)
   begin
   // Pull the pattern from MyString and place it into MySubString
   MySubString = mid(MyString, r.matchstart(),r.matchlength())
   // Use pushrow to create new rows - this is purely for the sake of
   // clarity in the example
   pushrow()
   // Create a while loop that continues to look for matches
   while r.findnext(MyString)
      begin
      // Pull the pattern from MyString and place it into MySubString agin
      MySubString = mid( MyString, r.matchstart(),r.matchlength())
      // Just for display again
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

MATCHLENGTH Function

Returns the length of the last pattern match found.

Category:	Regular Expression
Interaction:	This function operates on the pattern match substring found using FINDFIRST( ) or FINDNEXT( ).
Returned data type:	Integer

Syntax

r.**MATCHLENGTH**( )

Without Arguments

There is argument for this function.

Details

Use the MATCHLENGTH function to determine the length in characters of the currently matched pattern found with FINDFIRST( ) or FINDNEXT( ). Used in conjunction with

MATCHSTART( ), this function can be used to find matching substrings and populate variables in your expression code.

The function returns a positive integer value that represents the number of characters found to be a pattern match of the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some variables
integer i
string MyString

//Supply some values for the variables
i = 0
MyString = "DataFlux"
// Uncomment the line below to see the value of variable i change
//MyString = "Data_Management_Studio"

//You must define a regex object
regex r
//Then compile your regular expression.
// This expression will match as many "word" characters as it can
// (alphanumerics and undescore)
r.compile("\w*")
// If a pattern match is found then set i to show the length of
// the captured substring
if r.findfirst(MyString) then i = r.matchlength()
// Terminate the expression node processing
seteof()
```

MATCHSTART Function

Returns the location of the last pattern match found.

Category:	Regular Expression
Returned data type:	Integer

Syntax

r.**MATCHSTART**(*input*)

Required Argument

input
specifies a string value in which you want to search for the pattern defined by your compiled regular expression.

Requirement	*input* must not be NULL or blank.

Details

The MATCHSTART function returns the starting character position of a substring that has been matched to the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return. A logical loop can be used to iterate through all matching substrings. The MATCHLENGTH( ) function can be used in conjunction with MATCHSTART( ) to pull out matching substrings so that comparisons can be made to other values or to values stored in other variables.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
// Define some string variables
string MyString
string MySubString
integer StartLocation

// Set one to some sample input
MyString = "00AA111BBB2222CCCC"
// Will hold the starting location of matched patterns
StartLocation = 0

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]+")
// Find the first pattern match
if r.findfirst(MyString)
   begin
   // Pull the pattern from MyString and place it into MySubString
   MySubString = mid(MyString, r.matchstart(),r.matchlength())
   // Use pushrow to create new rows - this is purely for the sake of
   // clarity in the example
   pushrow()
   // Create a while loop that continues to look for matches
   while r.findnext(MyString)
      begin
      // Pull the pattern from MyString and place it into MySubString
      again
      MySubString = mid( MyString, r.matchstart(),r.matchlength())
      // Set StartLocation to the starting point of each pattern found
      StartLocation = r.matchstart()
      // Just for display again
      pushrow()
      end
   end
```

```
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

REPLACE Function

Searches for the first string, and replaces it with the second. This differs from the REPLACE() function used outside of the regex object.

Category:	Regular Expression
Returned data type:	String

Syntax

r.**REPLACE**(*input-string*, *replacement–value*)

Required Arguments

input-string
specifies a string value in which you want to search for and replaced in the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue"). This can also be a variable already defined in your expression code or passed to your expression node as a column from a previous node (MyValue or "My Value").

Requirement *input-string* must not be NULL or blank.

replacement–value
specifies a string to replace *input-string* that was matched by the compiled regular expression.

Details

The REPLACE function extends the capabilities of the regex object from simply finding patterns that match a regular expression to replacing the matching substring with a new value. For example, if you wanted to match all substrings that match a pattern of two hyphens with any letter in between (-A-, -B-) and replace with a single letter (Z), you would compile your regular expression for finding the hyphen/letter/hyphen pattern. Then you would use "Z" as the replacement value of the REPLACE() function passing in a variable or string value for input.

The return value is a string value with the replacement made if a replacement could indeed be made given the regular expression in play and the value supplied for input. If no replacement could be made, then the original value for input is returned

There are limitations to this functionality. You cannot easily replace the matched substring with a "captured" part of that substring. In the earlier example, you would have to parse the matched substring after it was found using FINDFIRST( ) or FINDNEXT( ) and create the replacement value based on that operation. But matched patterns can be of variable length, so guessing the position of parts of substrings can be tricky.

Compare this to similar functionality provided with using regular expressions as part of standardization definitions. In the case of QKB definitions that use regular expressions, much smarter replacements can be made because the regular expression engine enables you to use captured substrings in replacement values.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define two string variables
string MyString
string MyNewString

// Provide a value for MyString
MyString = "12Flux"

// Defined a regular expression object variable
regex r
// Compile a regular expression that will look for a series of digits
// either 2 or 3 digits long
r. compile("\d{2,3}")
// Use the replace function to place "Data" in place of the found
// pattern and save that in a new string variable.
// If you change MyString to 1234 or 12345 you can see the
// difference in how the pattern is found
MyNewString = r.replace(MyString,"Data")
// Terminate the expression node processing
seteof()
```

SUBSTRINGCOUNT Function

Returns the number of sub-patterns found to have matched the pattern specified by the compiled regular expression.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGCOUNT**( )

Without Arguments

There is no argument for this function.

Details

Use the SUBSTRINGCOUNT function to find the total number of sub-patterns found to have matched the regular expression. Normally simple regular expressions evaluate to

"1", but if you design regular expressions using sub-patterns then this function will return the number found.

The function returns a positive integer that specifies the number of substrings found to have matched the regular expression. A "0" is returned if no substrings are found.

The syntax for using subpatterns is open and closed parentheses. For example:

```
(Mr|Mrs) Smith
```

For this example, the sub-pattern is the "(Mr|Mrs)". Using this function returns the number "2" for the count of substrings since the entire string is considered the first sub-pattern. The part inside the parentheses is the second sub-pattern.

This function can provide the upper number for a logical loop using the FOR command so that your code can iterate through the matched sub-patterns for comparison to other values.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
      // Then pull out substrings
      SSS = r.substringstart(i)
```

```
        SSL = r.substringlength(i)
        MyString2 = mid(MyString,SSS,SSL)
        // Place the substrings in a data row
        pushrow()
        end
    end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGLENGTH

Returns the length of the n captured sub-pattern.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGLENGTH**(*n*)

Required Argument

n

specifies a positive integer that indicates the substring whose length you want to be returned.

Requirement	*n* must not be NULL or blank.

Details

The SUBSTRINGLENGTH function returns a positive integer value that represents the number of characters found to be a sub-pattern match of the regular expression. NULL is returned if there is no substring currently under consideration and therefore no length to return. For more information about working with sub-patterns, see the Details section of "SUBSTRINGCOUNT Function" on page 123.

Most simple regular expressions do not have sub-patterns, and this function behaves similarly to MATCHLENGTH( ). However, if your regular expression does use sub-patterns, then this function can be used to find the length of individually captured sub-patterns found within the overall matched pattern.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
      // Then pull out substrings
      SSS = r.substringstart(i)
      SSL = r.substringlength(i)
      MyString2 = mid(MyString,SSS,SSL)
      // Place the substrings in a data row
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGSTART Function

Returns the start location of the n captured sub-pattern.

Category:	Regular Expression
Requirement:	The regular expression must contain sub-patterns that can be used to match patterns.
Returned data type:	Integer

Syntax

r.**SUBSTRINGSTART**(*n*)

Required Argument

n

specifies a positive integer that indicates the sub-pattern whose starting location you want to be returned.

Requirement	*n* must not be NULL or blank.

Details

The SUBSTRINGSTART function takes the input integer *n* that you supply and returns a starting location for the sub-pattern represented by that input integer. NULL is returned if there is no substring currently under consideration and therefore no location to return.

Use SUBSTRINGCOUNT( ) to determine the number of sub-patterns under consideration. Use SUBSTRINGLENGTH( ) with this function to pull out the matched sub-patterns and use them in evaluation logic of your expression code.

Most simple regular expressions will not have sub-patterns and this function will behave similarly to MATCHSTART(). However, if your regular expression does use sub-patterns, then this function can be used to find the starting point of individually captured sub-patterns found within the overall matched pattern.

Example

Note: This example can be run in a stand-alone expression node if the **Generate rows when no parent is specified** option is selected. If passing data to this node instead, turn this setting off and remove the SETEOF() function. The PUSHROW statements are also unnecessary if passing data values in to the node as data rows. Unless stated otherwise, all code shown should be entered in the **Expression** tab of the **Expression** node.

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
```

```
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
   begin
      // Use the "substring" functions to find the number of substrings
      SSC = r.substringcount()
      // Loop through substrings
      for i = 1 to SSC
      begin
      // Then pull out substrings
      SSS = r.substringstart(i)
      SSL = r.substringlength(i)
      MyString2 = mid(MyString,SSS,SSL)
      // Place the substrings in a data row
      pushrow()
      end
   end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

Chapter 16
Search Functions

Overview

The following search function is available for the Expression Engine Language (EEL).

Dictionary

INLIST Function

Returns TRUE if the target parameter matches any of the value parameters.

Category:	Search
Returned data type:	Boolean

Syntax

INLIST(*target_parameter*, *value_parameter1* <, *value_parameter2*, …>)

Required Arguments

target_parameter
specifies a string, integer or date value to search.

value_parameter1
specifies a string, integer or date values to search for in *target_parameter*.

Optional Argument

***value_parameter2*, …**
specifies one or more string, integer or date values to search for in *target_parameter*.

Details

target_parameter is compared against each *value_parameter*. TRUE is returned if a match is found. Otherwise, FALSE is returned.

Example

```
string error_message

integer a
a=5
integer b
b=5

if (inlist(a,3,5)<>true)
   error_message="integer 5 not found in argument list of 3,5 "
else
   error_message="integer 5 was found in argument list of 3,5 "

print(error_message,false)
```

Chapter 17 String Functions

Overview

There are several functions available in Expression Engine Language (EEL) that affect the built-in string data type.

Dictionary

APARSE Function

Parses a string into words and returns the number of words found and places the words in an array.

Category:	String
Returned data type:	Integer

Syntax

APARSE(*string*, *delimiter*, *word_list*)

Required Arguments

string

specifies the string that needs to be separated into words; this can be specified as fixed string, field name, or expression

Restriction	*string* should not be NULL, it causes a run-time error.
Note	If *string* is empty (""), a value of 1 is returned and *word_list* has one element that contains an empty string.

delimiter

specifies a string that contains the character to be used as delimiter when separating the string into words; this can be specified as fixed string, field name, or expression

Restriction	If multiple characters are specified, only the last character is used.

word_list

specifies a string array that contains the words that were found during parsing; this is specified as a field name

Comparisons

The parse function is similar. It returns individual string fields instead of a string array, the string fields must be specified as part of the function invocation. The APARSE function does not have this restriction and can therefore be used when the maximum number of words is not known in advance.

Example

```
string = "one:two:three"
delimiter = ":"
nwords = aparse(string, delimiter, word_list) // outputs 3
first_word = word_list.get(1) // outputs "one"
last_word = word_list.get(nwords) // outputs "three"
```

ASC Function

Returns the position of a character in the ASCII collating sequence.

Category:	String
Returned data type:	Integer

Syntax

ASC(*string*)

Required Argument

string

specifies the character that needs to be found in the ASCII collating sequence; this can be specified as character constant, field name, or expression.

Restriction If multiple characters are specified, only the first character is used.

Details

See Appendix A: ASCII Values for a complete list of ASCII values.

Example

```
ascii_value = asc("a") // outputs 97
character_content = chr(97) // outputs the letter "a"
```

CHR Function

Returns an ASCII character for an ASCII code.

Category:	String
Returned data type:	String

Syntax

CHR(<*n*>)

Required Argument

n

specifies an integer that represents a specific ASCII character; this can be specified as a numeric constant, a field name, or an expression

Details

The CHR function returns nth character in the ASCII collating sequence.

Note: The CHR function can also be used to return any Unicode character when passed a Unicode code point.

See Appendix A: ASCII Values for a complete list of ASCII values.

Examples

Example 1

```
character_content = chr(97) // outputs the letter "a"
ascii_value = asc("a") // outputs 97
```

Example 2

The following examples support Unicode code points:

```
string input_string // this is a string that could contain greek characters
string(1) character
boolean greek_capital
for i=1 to len(input_string)
begin
   character=mid(input_string,i,1)
   if chr(character)>=913 and chr(character)<=939 then
      greek_capital=true
end
```

COMPARE Function

Returns the result of comparing two strings.

Category:	String
Returned data type:	Integer

Syntax

COMPARE(*string1*, *string2*<,*case-insensitive*>)

Required Arguments

string1
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

string2
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

Optional Argument

case-insensitive
specifies a Boolean string that indicates whether to compare case-insensitive strings; this can be specified as string constant, field name, or expression.

TRUE	specifies that the string comparison is not case sensitive.

FALSE	specifies that the comparison is case sensitive.

Default FALSE

Details

The MATCH_STRING function compares two strings lexicographically and can be used to do string comparisons using wildcards.

The return value is an integer representing the result of a lexicographical comparison of the two strings:

```
[-1 = string1 < string 2,
0 = string1 equals string2,
1 = string1 > string2]
```

To check whether two strings are equal, it is more efficient to use the == operator, for example:

```
if string1 == string2 then match=true
```

Example

```
// hallo comes before hello when alphabetically sorted
rc = compare("hello", "hallo" ) // outputs 1

// Hello comes before hello when alphabetically sorted
rc = compare("Hello", "hello" ) // outputs -1

modifier = null
rc = compare("Hello", "hello", modifier ) // outputs -1
rc = compare("Hello", "hello", true ) // outputs 0
```

EDIT_DISTANCE Function

Returns the number of corrections that would need to be applied to transform one string into the other.

Category:	String
Returned data type:	Integer

Syntax

EDIT_DISTANCE(*"string1"*, *"string2"***)**

Required Arguments

"string1"
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

"string2"
specifies a string to be used in the comparison; this can be specified as string constant, field name, or expression.

Details

The EDIT_DISTANCE function returns the number of corrections that need to be applied to transform *"string1"* into *"string2"*.

Example

```
distance = edit_distance("hello", "hllo" )
 // outputs 1

distance = edit_distance("hello", "hlelo" )
 // outputs 2

distance = edit_distance("hello", "hey" )
 // outputs 3
```

HAS_CONTROL_CHARS Function

Determines whether a string contains ASCII control characters.

Category:	String
Returned data type:	Boolean

Syntax

HAS_CONTROL_CHAR(*string*)

Required Argument

string
specifies a string to be search for the existence of ASCII control characters.

Details

The HAS_CONTROL_CHARS function can be used to identify non-printable ASCII control characters as found on the ASCII character table. A return value of TRUE indicates that control characters were found. A return value of FALSE indicates that control characters were not found.

Note: The only control character the HAS_CONTROL_CHARS function does not detect is 0 (null character).

See Appendix B: ASCII Control Characters for a list of control characters.

Example

```
boolean result_1

string error_message1

string test
test="Contol character: "&chr(13)
result_1=has_control_chars(test)
```

```
if(result_1)
   error_message1 = "test string contains control character"
else
   error_message1 = "test string does not contain control character"
```

INSTR Function

Returns the position of one string within another string.

Category:	String
Returned data type:	Integer

Syntax

INSTR(*source*, *excerpt*<, *count*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

excerpt
specifies a string to search for within *source*; this can be specified as string constant, field name, or expression.

Optional Argument

count
specifies an integer that indicates the occurrence of *excerpt* to search for; this can be specified as numeric constant, field name, or expression. For example, a value of 2 indicates to search for the second occurrence of *excerpt* in *source*.

Details

The INSTR function searches *source* from left to right for the *count*-th occurrence of *excerpt*. If *excerpt* is not found in *source*, the function returns a value of 0.

Example

```
source = "This is a simple sentence."
excerpt = "is"

position = instr(source, excerpt, 1) // outputs 3
position = instr(source, excerpt, 2) // outputs 6
```

LEFT Function

Returns the left-most characters of a string.

Category:	String

Returned data type: String

Syntax

LEFT(*source*, *count*)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

count
specifies an integer that indicates how many characters to return; this can be specified as numeric constant, field name, or expression.

Note When a count of zero or less is specified, an empty string is returned.

Example

```
source = "abcdefg"
result = left(source, 4) // outputs the string "abcd"
```

LEN Function

Returns the length of a string.

Category: String

Returned data type: Integer

Syntax

LEN(*source*)

Required Argument

source
specifies a string for which the length needs to be determined; this can be specified as string constant, field name, or expression.

Note The length of an empty string ("") is zero. If *source* is NULL, the function returns a NULL value.

Tip To remove leading and trailing blanks, use the trim function.

Example

```
string(30) source
```

```
source = "abcdefg"
length_string = len(source) // outputs 7

source = "  abcdefg  "
length_string = len(source) // outputs 11

source = " " // source contains a blank
length_string = len(source) // outputs 1
```

LOWER Function

Converts a string to lowercase.

Category: String

Returned data type: String

Syntax

LOWER(*source*)

Required Argument

source
specifies a string; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

Example

```
source = "MÜNCHEN in Germany"
lowcase_string = lower(source) // outputs "münchen in germany"
```

MATCH_STRING Function

Determines whether the first string matches the second string, which can contain wildcards.

Category: String

Returned data type: Boolean

Syntax

MATCH_STRING(*string1*, *string2*)

Required Arguments

string1
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL the function returns a NULL value.

string2
specifies a string that represents a search pattern; this can be specified as string constant, field name, or expression.

Details

The MATCH_STRING function searches *string1* using the search pattern specified in *string2*. If a match was found, true is returned. Otherwise, false is returned.

Search strings can include wildcards in the leading (*ABC) and trailing (ABC*) position, or a combination of the two (*ABC*). Wildcards within a string are invalid (A*BC).

A question mark can be used as a wildcard but is matched only to a character. For example, AB? will match ABC, not AB.

To execute a search for a character that is used as a wildcard, precede the character with a backslash. This denotes that the character should be used literally and not as a wildcard. Valid search strings include: *BCD*, *B?D*, *BCDE, *BC?E, *BCD?, ABCD*, AB?D*, ?BCD*, *B??*, *B\?\\* (will match the literal string AB?\E). An invalid example is: AB*DE.

For more complex searches, use regular expressions instead of the MATCH_STRING() function.

Example

```
string1 = "Monday is sunny, Tuesday is rainy & Wednesday is windy"
string2 = "Tuesday is"
match = match_string(string1, string2) // outputs false
string2 = "*Tuesday is*"
match = match_string(string1, string2) // outputs true
```

MID Function

Extracts a substring from an argument.

Category:	String
Returned data type:	String

Syntax

MID(*source*, *position*<, *length*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

position
specifies an integer that is the beginning character position; this can be specified as a numeric constant, field name, or expression.

Optional Argument

length
specifies an integer that is the length of the substring to extract; this can be specified as a numeric constant, field name, or expression.

Default If length is omitted, the remainder of the string will be extracted.

Note If length is NULL, zero, or larger than the length of the expression that remains in source after position, the remainder of the expression is returned.

Example

```
source = "06MAY15"

result = mid(source, 3, 3) // outputs "MAY"
result = mid(source, 3) // outputs "MAY15"
```

PARSE Function

Parses a string into words and returns the number of words found and the words found.

Category:	String
Returned data type:	Integer

Syntax

PARSE(*string*, *delimiter*, *word1*<, *word2* , …>)

Required Arguments

string
specifies a string with delimiters that is to be separated into words; this can be specified as fixed string, field name, or expression.

Note If *string* is NULL, the function returns a NULL value. If *string* is empty ("") a value of 1 is returned.

delimiter
specifies a character that delimits the words in a string; this can be specified as fixed string, field name, or expression.

word1
specifies a string that represents the first word found; this is specified as field names.

Optional Argument

word2, …
specifies one or more strings that represents, in order, strings that are found after the first string; these are specified as field names.

Details

The parse function assigns to the provided parameters the words found in *string* that are separated by a delimiter. The return values indicates the number of words found.

Comparisons

The APARSE function is similar. The APARSE function is more flexible as you do not have to know in advance the maximum number of words. It can also be used to easily determine the last word in a string.

Example

```
string = "one:two:three"
delimiter = ":"
nwords = parse(string, delimiter, word1, word2) // outputs 3
// word1 will contain the value "one"
// word2 will contain the value "two"
```

PATTERN Function

Indicates whether a string has numbers, uppercase characters, and lowercase characters.

Category:	String
Returned data type:	String

Syntax

PATTERN(*string*)

Required Argument

string
specifies a string that is to be evaluated for numbers, uppercase characters, and lowercase characters; this can be specified as fixed string, field name, or expression

Note If string is NULL, the function returns a NULL value. If string is empty (""), an empty value is returned.

Details

The returned string contains a 9 in place of each number, an “A” for each uppercase character, and an “a” for each lowercase character. Other characters are not replaced.

Example

```
source_string = "12/b Abc-Str."
result = pattern(source_string) // outputs "99/a Aaa-Aaa."
```

REPLACE Function

Replaces the first occurrence of one string with another string, and returns the resulting string.

Category:	String
Returned data type:	String

Syntax

REPLACE(*source*, *search-string*, *replace-string*<, *count*>)

Required Arguments

source
specifies a string to search; this can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

search-string
specifies the text that is to be replaced; this can be specified as string constant, field name, or expression.

replace-string
specifies a string that is to replace *search-string*; this can be specified as string constant, field name, or expression.

Optional Argument

count
specifies an integer that represents how many replacements should be made; this can be specified as numeric constant, field name, or expression.

Note If count is omitted or set to zero, all occurrences will be replaced in the string.

Example

```
source_string =
     "It's a first! This is the first time I came in first place!"
search = "first"
replace = "second"
count = 2
result = replace(source_string, search, replace, count)
// outputs "It's a second! This is the second time I came in first place!"
```

RIGHT Function

Returns the right-most characters of a string.

Category:	String
Returned data type:	String

Syntax

RIGHT(*source*, *count*)

Required Arguments

source
specifies a string to be searched; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

count
specifies an integer that indicates how many characters to return; this can be specified as numeric constant, field name, or expression.

Note When *count* is zero or less, an empty string is returned.

Example

```
source = "abcdefg"
result = right(source, 4) // outputs the string "defg"

source = "abcdefg "
result = right(source, 2) // outputs the string "fg "
```

SORT Function

Returns a string with its characters sorted alphabetically.

Category:	String
Returned data type:	String

Syntax

SORT(*source* <, *ascending*, *remove_duplicates*>)

Required Argument

source

specifies a string to sort; this can be specified as string constant, field name, or expression.

Note If *source* is NULL, the function returns a NULL value.

Optional Arguments

ascending

specifies whether the text should be sorted in ascending order; this can be specified as Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE	the input string is sorted in ascending order.
FALSE	the input string is sorted in descending order.

Default TRUE

remove_duplicates

specifies whether duplicate characters should be removed; this can be specified as Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE	duplicate characters are removed.
FALSE	duplicate characters are not removed.

Default FALSE

Details

In determining the order, special characters precede initial capital letters, which precede lowercase letters.

Example

```
source_string = "A short Sentence."
ascending = true
remove_duplicates = true
result = sort(source_string, ascending, remove_duplicates)
// outputs ".AScehnorst"
```

SORT_WORDS Function

Returns a string that consists of the words that are sorted alphabetically.

Category:	String
Returned data type:	String
Note:	Special characters such as ",.!" are not treated as separation characters.

Syntax

SORT_WORDS(*source*<,*ascending*,*remove_duplicates*>)

Required Argument

source
specifies a string to sort; the string can be specified as string constant, field name, or expression.

Note If source is NULL, the function returns a NULL value.

Optional Arguments

ascending
specifies whether the words in the input string should be sorted in ascending order; this can be specified as a Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE	the input string is sorted in ascending order.
FALSE	the input string is sorted in descending order.

Default TRUE

remove_duplicates
specifies whether duplicate words should be removed; this can be specified as a Boolean constant, field name, or expression. The value must evaluate to either TRUE or FALSE:

TRUE	duplicate words are removed.
FALSE	duplicate words are not removed.

Default FALSE

Details

In determining the order, words with initial capital letters precede lowercase letters. Also, words with a concatenated special character are treated as a different word. For example, first and first! are two different words.

Example

```
source_string =
     "It's a first! This is the first time I came in first place!"
ascending = true
remove_duplicates = true
result = sort_words(source_string, ascending, remove_duplicates)
// outputs "I It's This a came first first! in is place! the time"
```

TODATE Function

Converts the argument to a date value based on the locale settings on your system.

Category: String

Interaction: The TODATE( ) function depends on the default **Short date** regional setting on your Microsoft Windows environment. You can change the locale setting on Windows.

Returned data type: Date

Syntax

TODATE(*any*)

Required Argument

any
specifies a value to convert to a date.

Details

When the TODATE( ) function is evaluated, the value of the Windows **Short date** locale field is examined to determine whether the format is MM/DD/YYYY or DD/MM/YYYY. The order of the month and date is determined by the **Short date** value.

Example

```
// Declare the date variable to contain the date value
date dateval

// Use the TODATE function to populate the date variable
dateval=todate(3750)

// Returns the value:
4/7/10 12:00:00 AM
```

TOINTEGER Function

Converts the argument to an integer value.

Category: String

Returned data type: Integer

Syntax

TOINTEGER(*any*)

Required Argument

any
specifies a value to convert to an integer.

Examples

Example 1

```
if tointeger(counter)<= 5
setoutputslot(0)
else
setoutputslot(1)
```

Example 2

```
// Declare an integer variable to contain the integer value
integer intval

// Use the TOINTEGER function to populate the integer variable
intval=tointeger(3750.12345)

// Returns the value:
3750
```

See Also

For rules and special considerations when you coerce types, see “Coercion” on page 15.

TOREAL Function

Converts the argument to a real value.

Category:	String
Returned data type:	Real

Syntax

TOREAL(*any*)

Required Argument

any
specifies the value that is to be converted to a real value.

Example

```
// Declare a real variable to contain the real value
real realval

// Use the TOREAL function to populate the real variable
realval=toreal(3750.12345)

// Returns the value:
3750.12345
```

See Also

For rules and special considerations when you coerce types, see "Coercion" on page 15.

TOSTRING Function

Converts the argument to a string value.

Category:	String
Returned data type:	String

Syntax

TOSTRING(*any*)

Required Argument

any
specifies any non-string value to conversion to a string.

Example

```
// Declare a string variable to contain the string
String stringval

// Use the TOINTEGER function to populate the integer variable
stringval=tostring(3750.12345)

// Returns the string
3750.12345
```

See Also

For rules and special considerations when you coerce types, see "Coercion" on page 15.

TRIM Function

Removes leading and trailing white space.

Category:	String
Returned data type:	String

Syntax

TRIM(*source*)

Required Argument

source
a string from which the leading and trailing white space needs to be removed; this can be specified as string constant, s field name, or an expression.

Note If source is NULL, the function returns a NULL value. If source is an empty value (""), the function returns an empty value.

Example

```
source = " abcd " // 2 leading and 2 trailing spaces
result = trim(source) // outputs "abcd"
length = len(source) // outputs 8
length = len(result) // outputs 4
```

UPPER Function

Converts a string to uppercase.

Category:	String
Returned data type:	String

Syntax

UPPER(*source*)

Required Argument

source
specifies a string constant, a field name, or an expression.

Note If *source* is NULL, the function returns a NULL value.

Example

```
source = "MÜNCHEN in Germany"
upcase_string = upper(source) // outputs "MÜNCHEN IN GERMANY"
```

USERNAME Function

In DataFlux Data Management Studio, the USERNAME function returns the user name of the user that is logged in to the operating system. However, in DataFlux Data Management Server the USERNAME function returns the account name for the Data Management Server process.

Category:	String
Returned data type:	String

Syntax

USERNAME ( )

Without Arguments

This function does not take arguments.

Example

```
// Declare the string value for the function
string user

// For Data Management Studio, use the USERNAME
// function to get the operating system user name
user = username()

// For Data Management Server, use the USERNAME
// function to get the account name for the Data
// Management Server process.
user = username()
```

Part 3

Appendixes

Appendix 1 Frequently Asked Questions

This section introduces frequently asked questions along with exercises. These topics include examples that illustrate specific concepts related to the Expression Engine Language (EEL).

Testing and Evaluating

In order to test an expression before running a data job, you must create sample rows.

Exercise 1: How do I test an expression without using a table to create rows?

In the Expression Properties dialog box, select **Generate rows when no parent is specified**.

This creates sample empty rows in the Preview tab.

Note: If you do not select **Generate rows when no parent is specified**, and you do not have output specified in the post-processing step, no data is output.

Exercise 2: Is it possible to create test rows with content rather than empty rows?

This involves creating extra rows with the PUSHROW() function in the Pre-expression section.

Note: To use the PUSHROW() function, do not select **Generate rows when no parent is specified**.

Consider the code example below:

```
// Pre-Expression
string name // the name of the person
string address // the address of the person
integer age // the age of the person

// Content for the first row
name="Bob"
address="106 NorthWoods Village Dr"
age=30

 // Create an extra row for the
// fields defined above
pushrow()

// The content for the extra row
name="Adam"
```

```
address="100 RhineStone Circle"
age=32

// Create an extra row for the
// fields defined above
pushrow()

// The content for extra row
name="Mary"
address="105 Liles Rd"
age=28

// Create an extra row for the
// fields defined above
pushrow()
```

The PUSHROW() function creates the rows.

Selecting Output Fields

Some fields are used for calculation or to contain intermediate values, but are not meaningful in the output. As you test or build scripts, you might need to exclude fields from the output.

Exercise: How do I exclude some fields in the expression from being listed in the output?

To accomplish this, use the keyword hidden before declaring a variable.

Consider the following example:

```
// Pre-Expression
// This declares a string
// type that will be hidden
hidden string noDisplay

// Expression
// Assigns any value to the string type
noDisplay='Hello World But Hidden'
```

The noDisplay string field is not output to the Preview tab.

To verify this, remove the parameter hidden from the string noDisplay declaration. Observe that noDisplay is output.

Working with Subsets

When you work with large record sets in the Data Job Editor, it can be time-consuming to test new jobs. You can shorten this time when you build your expression and test your logic against a subset of large record sets.

Exercise 1: Apply Your Expression to a Subset of Your Data By Controlling the Number of Records Processed.

Consider the following example:

```
// Pre-Expression

// We make this variable hidden so it is
// not output to the screen
hidden integer count

count=0
hidden integer subset_num

// the size of the subnet
subset_num=100

// This function estimates and sets the # of
// records that this step will report
rowestimate(subset_num)

// Expression
if(count==subset_num)
     seteof()
else
     count=count + 1
```

Keep track of the number of records output with the integer variable count. Once count matches the size of the subset, use the SETEOF() function to prevent any more rows from being created.

The exact syntax for SETEOF() function is:

```
boolean seteof(boolean)
```

When SETEOF() is called, the node does not read any more rows from the parent node. If **Generate rows when no parent is specified** is checked, the node stops generating rows. Furthermore, if any rows have been pushed using PUSHROW(), they are discarded, and further calls to PUSHROW() have no effect. The exception to this is if SETEOF(true) is called. In this case, any pushed rows (whether pushed before or after the call to SETEOF()) are still returned to the node below. Notably, if further PUSHROW() calls occur after SETEOF(true) is called, these rows are returned as well. Also note that after SETEOF() is called, the post-group expression and the post expression are still executed.

The ROWESTIMATE() function is used by data jobs to estimate the number of records that will be output from this step.

If you remove the hidden parameter from the integer count declaration, integers 1–100 are output.

Another approach to solving this problem is shown in the following example:

Exercise 2: Apply Your Expression to a Subset of Your Code By Filtering Out Rows of Data.

By setting the return value to true or false, you can use this approach as a filter to select which rows go to the next step.

Note: If you always return false, you get no output and your expression enters an infinite loop.

```
// Pre-Expression
integer counter
counter=0
integer subset_num

subset_num=50
```

```
// Expression
if counter < subset_num
     begin
         counter=counter + 1
     end
else
     return true
```

Initializing and Declaring Variables

As an expression is evaluated, each row updates with the values of the fields in the expression. This can lead to re-initialization of certain variables in the expression. You might want to initialize a variable only once and then use its value for the rest of the expression script.

Exercise: How do I initialize a variable just one time and not with each iteration of a loop?

Declare the variable in the pre-expression step, and it is initialized only once before the expression process takes over.

Saving Expressions

The following exercise explains the steps required to save your expressions.

Exercise: How do I save my expressions?

You can save an expression without saving the entire data job. Click Save. Your expression is saved in an .exp text file format that you can load using Load.

Counting Records

The following exercises explain expressions used for record count.

Exercise 1: How do I count the number of records in a table using the EEL?

In this example, a connection is made to the Contacts table in the DataFlux sample database, and output to an HTML report. For more information about connecting to a data source and specifying data outputs, refer to the DataFlux Data Management Studio online Help.

Define an integer type in the pre-expression step that contains the count.

```
// Pre-Expression
// Declare and initialize an integer
// variable for the record count
integer recordCount
recordCount=0

// Expression
// Increment recordCount by one
recordCount=recordCount+1
```

The value of RECORDCOUNT increases in increments of one until the final count is reached. If you want to increase the count for only those values that do not have a null value, enter the following in the expression:

```
// Check if the value is null
if(NOT isnull(`address`) ) then
     recordCount=recordCount+1
```

In this example, the value RECORDCOUNT is updated after each row iteration.

Note: Field names must be enclosed in grave accents (ASCII `) rather than apostrophes (ASCII ').

Exercise 2: How do I see the final count of the records in a table instead of seeing it incremented by one on every row?

Declare a count variable as hidden. In the post-expression step, assign the value of count to another field that you want to display in the output (FINALCOUNT). Using PUSHROW(), add an extra row to the output to display FINALCOUNT. Add the final row in the post-processing step, so that FINALCOUNT is assigned only after all of the rows are processed in the expression step.

Here is the EEL code:

```
// Preprocessing hidden integer count count=0

// Expression
if(NOT isnull(`address`) ) then
     count=count+1

// Post Processing
// Create a variable that will contain
// the final value and assign it a value
integer finalCount

finalCount=count

// Add an extra row to the output
pushrow()
```

When you enter this code and then run the code, the last row should display the total number of records in the table that are not null.

Exercise 3: How do I get just one row in the end with the final count instead of browsing through a number of rows until I come to the last one?

A simple way to do this is to return false from the main expression. The only row that is output is the one that was created with PUSHROW().

Or you can devise a way to indicate that a row is being pushed. The final row displayed is an extra pushed row on top of the stack of rows that is displayed. Therefore, you can filter all the other rows from your view except the pushed row.

To indicate that a row is pushed on your expression step, select **Pushed status field** and enter a new name for the field.

Once you indicate with a Boolean field whether a row is pushed or not, add another expression step that filters rows that are not pushed:

```
// Preprocessing
hidden integer count
count=0

// Add a boolean field to indicate
// if the row is pushed
boolean pushed

// Expression
if(NOT isnull(`address`) ) then
     count=count+1

// Name the pushed status field "pushed"
if (pushed) then
     return true
else
     return false

// Post Processing
integer finalCount
finalCount=count

pushrow()
```

Debugging and Printing Error Messages

The following exercise explains how to print error messages and find debugging information.

Exercise: Is there a way to print error messages or to get debugging information?

You can use the PRINT() function that is available to print messages. When previewing output, these messages print to the Log tab.

In a previous example of calculating the number of records in a table, in the end, you can output the final count to the statistics file. In the post-processing section, you would use the following.

```
// Post Processing

// Integer to have the final count
integer finalCount

finalCount=count

// Add one extra row for post processing
pushrow()

// Print result to file
print('The final value for count is: '& finalCount)
```

Creating Groups

Expressions provide the ability to organize content into groups. The EEL has built-in grouping functionality that contains this logic. Once data is grouped, you can use other functions to perform actions on the grouped data. The use of grouping in EEL is similar to the use of the Group By clause in SQL.

Exercise 1: Can EEL group my data and then count the number of times each different entry occurs?

Yes. For example, you can count the number of different states that contacts are coming from, using the contacts table from a DataFlux sample database.

Exercise 2: How can I count each state in the input so that "NC", "North Carolina", and "N Carolina" are grouped together?

A convenient way to accomplish this is to add an expression node or a standardization node in the Data Job Editor, where you can standardize all entries prior to grouping.

Building on the previous example, add a Standardization step:

1. In Data Job Editor, click Quality.
2. Double-click the Standardization node.
3. In the Standardization Properties dialog box, select **State** and specify the State/Province (Abbreviation) definition. This creates a new field called STATE_Stnd.
4. Click **Additional Outputs** and select **All**.
5. Click **OK**.
6. In the Standardization Properties dialog box, click **OK**.
7. In the Expression Properties dialog box, click **Grouping**. Make sure that grouping is now by STATE_Stnd and not STATE.
8. Click **OK**.

The STATECOUNT now increments by each standardized state name rather than by each permutation of state and province names.

Exercise 3: How do I group my data and find averages for each group?

To illustrate how this can be done, use sample data.

1. Connect to a Data Source.
   a. Connect to the Purchase table in the DataFlux sample database.
   b. In the Data Source Properties dialog box, click **Add All**.
   c. Find the Field Name for ITEM AMOUNT. Change the Output Name to ITEM_AMOUNT.
2. Sort the Data. Now that you have connected to the Purchase table, sort on the data field that you use for grouping. In this case, sort by DEPARTMENT.
   a. In the Data Job Editor, click **Utilities**.
   b. Double-click **Data Sorting**. This adds a Data Sorting node.
   c. In the Data Sorting Properties dialog box, select **DEPARTMENT** and set the Sort Order to Ascending.
   d. Click **OK**.
3. Create Groups. To create groups out of the incoming data, add another Expression node to the job after the sorting step.
   a. In the Expression Properties dialog box, click **Grouping**. The following three tabs are displayed: Group Fields, Group Pre-Expression, and Group Post-Expression.

b. On the Group Fields tab, select **DEPARTMENT**.

c. On the Group Pre-Expression tab, declare the following fields, and then click **OK**:

```
// Group Pre-Expression
// This variable will contain the total
// sales per department
real total
total=0

// This variable will keep track of the
// number of records for each department
integer count
count=0

// This variable will contain the
// running average total
real average
average=0
```

d. On the Expression tab, update the variables with each upcoming new row, and then click **OK**:

```
// Expression
// increase the total sales
total=total+ITEM_AMOUNT

// increase the number of entries
count=count+1

// error checking that the count of entries is not 0
if count !=0 then
     begin
          average=total/count
          average=round(average,2)
     end
```

When you preview the Expression node, you should see the following in the last four columns:

Department	Total	Count	Average
1	3791.7	1	3791.7
1	6025.4	2	3012.7
1	7294.5	3	2431.5
1	11155.2	4	2788.8
...	...	...	...

Retrieving and Converting Binary Data

EEL provides the ability to retrieve data in binary format. This section describes how to retrieve and convert binary data in big-endian or little-endian formats, as well as mainframe and packed data formats.

Exercise 1: How do I retrieve binary data in either big-endian or little-endian format?

To retrieve binary data, use the IB() function. It also determines the byte order based on your host or native system. The syntax is:

```
real = ib(string, format_str)
```

where:

- string: The octet array containing binary data to convert.
- format_str: The string containing the format of the data, expressed as w.d. The width (w) must be between 1 and 8, inclusive, with the default as 4. The optional decimal (d) must be between 0 and 10, inclusive.

The w.d formats and informats specify the width of the data in bytes. The optional decimal portion specifies an integer which represents the power of ten by which to divide (when reading) or multiply (when formatting) the data. For example:

```
//Expression
//File handler to open the binary file
file input_file
//The binary value to be retrieved
real value
//The number of bytes that were read
integer bytes_read
//4-byte string buffer
string(4) buffer
input_file.open("C:\binary_file", "r")
//This reads the 4 byte string buffer
bytes_read=input_file.readbytes(4, buffer)
//The width (4) specifies 4 bytes read
//The decimal (0) specifies that the data is not divided by any power of ten
value = ib(buffer,"4.0")
```

Exercise 2: How do I force my system to read big-endian data regardless of its endianness?

To force your system to read big-endian data, use the S370FIB() function. The syntax is:

```
real = s370fib(string, format_str)
```

where:

- string: The octet array containing IBM mainframe binary data to convert.
- format_str: The string containing the w.d format of the data.

Use this function just like the IB() function. This function always reads binary data in big-endian format. The S370FIB() function is incorporated for reading IBM mainframe binary data.

Exercise 3: How do I read little-endian data regardless of the endianness of my system?

Currently, there are no functions available for this purpose.

Exercise 4: How do I read IBM mainframe binary data?

To read IBM mainframe binary data, use the S370FIB() function, described in Exercise 2.

Exercise 5: How do I read binary data on other non-IBM mainframes?

Currently, there are no functions available for this purpose.

Exercise 6: Is there support for reading binary packed data on IBM mainframes?

To read binary packed data on IBM mainframes, use the function S370FPD(). The syntax is:

```
real = s370fpd(string, format_str)
```

where:

- string: The octet array containing IBM mainframe packed decimal data to convert.
- format_str: The string containing the w.d format of the data.

This function retrieves IBM mainframe-packed decimal values. The width (w) must be between 1 and 16, inclusive, with the default as 1. The optional decimal (d) must be between 0 and 10, inclusive. This function treats your data in big-endian format.

Exercise 7: How do I read non-IBM mainframe packed data?

To read non-IBM mainframe packed data, use the function PD(). The syntax is:

```
real = pd(string, format_str)
```

where:

- string: The octet array containing IBM mainframe binary data to convert.
- format_str: The string containing the w.d format of the data.

Converting Binary Data to a Certain Format

Just as it is possible to retrieve data in a special binary format, it is also possible to format data to a special binary format.

Exercise 8: How do I format binary data to the native endianness of my system?

To format binary data, use the FORMATIB() function. The syntax is:

```
integer = formatib(real, format_str, string)
```

where:

- real: The numeric to convert to a native endian binary value.
- format_str: The string containing the w.d format of the data.
- string: The octet array in which to place the formatted native endian binary data. returns:
- integer: The byte length of formatted binary data.

This function produces native endian integer binary values. The width (w) must be between 1 and 8, inclusive, with the default as 4. For example:

```
//Expression
//The byte size of the buffer that contains the content
real format_size
//The real type number
real number
//The real number that is retrieved
real fib_format
number=10.125
//The buffer that contains the formatted data
string(4) buffer
format_size= formatib(number, "4.3", buffer
//4.3 is to specify 4 bytes to read the entire
//data and 3 to multiply it by 1000
//The reason to multiply it by a 1000 is to divide it later by 1000
//To restore it back to a real number
fib_format= ib(buffer, "4.3")
//Verify that the formatting worked
//Fib_format should be 10.125
```

Exercise 9: How do I change to other formats?

To change to other formats, use the following functions:

Non-IBM mainframe packed data

```
integer = formatpd(real, format_str, string)
```

where:

- real: The numeric to convert to a native-packed decimal value.
- format_str: The string containing the w.d format of the data.
- string: The octet array in which to place the formatted native-packed decimal data.

returns:

- integer: The byte length of formatted packed decimal data.

IBM mainframe binary data

```
integer = formats370fib(real, format_str, string)
```

where:

- real: The numeric to convert to an IBM Mainframe binary value.
- format_str: The string containing the w.d format of the data.
- string: The octet array in which to place the formatted IBM mainframe binary data.

returns:

- integer: The byte length of formatted binary data

IBM mainframe packed decimal data

```
integer = formats370fpd(real, format_str, string)
```

where:

- real: The numeric to convert to an IBM mainframe-packed decimal value.
- format_str: The string containing the w.d format of the data.

- string: The octet array in which to place the formatted IBM mainframe-packed decimal data.

returns:

- integer: The byte length of formatted packed decimal data.

Supporting COBOL

Using expressions, it is possible to read binary data in specified COBOL COMP, COMP-3, and COMP-5 data formats. The following examples demonstrate how to do this.

Exercise 1: How do I read native endian binary data for COBOL?

To read native endian binary data, use the PICCOMP() function. The syntax is:

```
real piccomp(string, format_str)
```

where:

- string: The octet array containing COBOL formatted packed decimal data to convert.
- format_str: The string containing the PIC 9 format of the data.

The PICCOMP() function determines the number of bytes (2, 4, or 8) to consume by comparing the sum of the 9s in the integer and fraction portions to fixed ranges. If the sum is less than 5, then 2 bytes are consumed. If the sum is greater than 4 and less than 10, then 4 bytes are consumed. If the sum is greater than 9 and less than 19, then 8 bytes are consumed. For example:

```
//Expression
//file handler to open files
File pd
integer rc
string(4) buffer
real comp
if (pd.open("binary_input.out", "r")) begin
rc = pd.readbytes(4, buffer)
if (4 == rc) then
comp = piccomp(buffer, "S9(8)")
pd.close()
end
```

In the preceding example, because of the format of the string is S9(8), 4 bytes were consumed. Notice that all of the COBOL data functions support a PIC designator of the long form:

[S][9+][V9+] (ex: S99999, 99999V99, S999999V99, SV99)

Or of the shortened count form:

[S][9(count)][V9(count)] (ex: S9(5), 9(5)v99, S9(6)v9(2), sv9(2))

Exercise 2: How do I read packed decimal numbers?

To read packed decimal numbers, use the PICCOMP3() function. The syntax is:

```
real piccomp3(string, format_str)
```

where:

- string: The octet array containing COBOL-formatted packed decimal data to convert.

- format_str: The string containing the PIC 9 format of the data.

The PICCOMP3() function determines the number of bytes to consume by taking the sum of the 9s in the integer and fraction portions and adding 1. If the new value is odd, 1 is added to make it even. The result is then divided by 2. As such, S9(7) would mean there are 4 bytes to consume. The packed data is always in big-endian form.

The PICCOMP3() function is used the same as the PICCOMP() function. For an example of the PICCOMP3() function, see Exercise 1.

Exercise 3: How do I read signed decimal numbers in COBOL format?

To read signed decimal numbers, use the PICSIGNDEC() function. The syntax is:

```
real picsigndec(string buffer, string format_str, boolean ebcdic, boolean trailing)
```

where:

- string buffer: The octet array containing a COBOL-formatted signed decimal number to convert.
- string format_str: The string containing the PIC 9 format of the data. The default format_str is S9(4).
- Boolean EBCDIC: The Boolean when set to nonzero indicates the string is EBCDIC. The default EBCDIC setting is false.
- Boolean trailing: The Boolean when set to nonzero indicates the sign is trailing. The default trailing setting is true.

The PICSIGNDEC() function determines the number of bytes to consume by taking the sum of the 9s in the integer and fraction portions of format_str. For example:

```
//Expression
//file handler to open files
file pd
integer rc
string(6) buffer
real comp
if (pd.open("binary_input.out", "r")) begin
rc = pd.readbytes(6, buffer)
if (4 == rc) then
comp = picsigndec(buffer, "S9(4)V99",1,1) pd.close()
end
```

Formatting

It is also possible to format data to a specific COBOL format, as demonstrated by the following exercises:

Exercise 4: How do I format from a real to COBOL format?

To format from a real to a COBOL format, use the FORMATPICCOMP() function. The syntax is:

```
integer = formatpiccomp(Real number,string format_str, string result)
```

where:

- real number: The numeric to convert to a COBOL native endian binary value.
- string format_str: The string containing the PIC 9 format of the data.

- string result: The octet array in which to place the COBOL-formatted native endian binary data.

returns:

- integer: The byte length of formatted binary data.

The FORMATPICCOMP() function does the reverse of PICCOMP(). As with the PICSIGNDEC() function, the FORMATPICSIGNDEC() function determines the number of bytes to consume by taking the sum of the 9s in the integer and fraction portions. For example:

```
//Expression
real comp
comp = 10.125
integer rc
rc = formatpiccomp(comp, "s99V999", buffer)
//The string buffer will contain the real value comp formatted to
platform COBOL COMP native endian format. ??///
```

Exercise 5: What is the list of functions available for COBOL formatting?

The syntax for a COBOL-packed decimal value is:

```
integer = formatpiccomp3(Real number, string format_str, string result)
```

where:

- real number: The numeric to convert to a COBOL packed decimal value.
- string format_str: The string containing the PIC 9 format of the data.
- string result: The octet array in which to place the COBOL formatted packed decimal data.

returns:

- integer: The byte length of formatted packed decimal data.

The syntax for a COBOL-signed decimal value is:

```
integer = formatpicsigndec(real number, string format_str, string buffer,
boolean ebcdic, boolean trailing)
```

where:

- real number: The numeric to convert to a COBOL-signed decimal value.
- string format_str: The string containing the PIC 9 format of the data.
- string buffer: The octet array in which to place the COBOL-formatted packed decimal data.
- Boolean EBCDIC: The Boolean when nonzero indicates to format in EBCDIC.
- Boolean trailing: The Boolean when nonzero indicates to set the sign on the trailing byte.

returns:

- integer: The byte length of the formatted signed decimal.

The COBOL-format functions are used the same as the FORMATPICCOMP() function. For an example of the COBOL-format functions, see Exercise 4.

Using Array Functions

This section contains additional information about arrays, including:

Creating an Array

The following exercise explains how to create an array.

Exercise: How do I create an array and provide values for the items in the array?

To declare an array, use the reserved key word array.

```
string array variable_name
integer array variable_name
boolean array variable_name
date array variable_name
real array variable_name
```

For example:

```
// declare an array of integer types
integer array integer_list

// set the size of the array to 5 integer_list.dim(5)

// the index that will go through the array
integer index
index=0

// Set the values of the items inside the
// array to their index number
for index=1 to 5
    begin
        integer_list.set(index, index);
    end
```

Retrieving Elements from an Array

This exercise demonstrates how to retrieve elements from an array.

Exercise: How do I retrieve elements from an array?

To retrieve elements from an array, use the following example; it builds on the previous example:

```
integer first
integer last

// Getting the first item from integer array
first=integer_list.get(1);
// Getting the last item from integer array
last=integer_list.get(5)
```

Changing an Array Size

The following exercise explains how to change the size of an array.

Exercise: How do I change the size of an array?

To change the size of an array, use the DIM() function. For example:

```
// array is originally initialized to 5
string array string_container
string_container.dim(5)
...
...
// the array is sized now to 10
string_container.dim(10)
```

Determining an Array's Size

The following exercise shows how to determine the size of an array.

Exercise: How do I determine the size of an array?

To determine the size of an array, use the DIM() function. Remember that the DIM() function is also used to set the size of an array. If no parameter is specified, the array size does not change.

For example:

```
// Expression
integer array_size string array array_lister
...
...

// after performing some operations on the array
// array_size will then contain
// the size of the array
array_size=array_lister.dim()
```

Finding Common Values between Columns Using Arrays

The next exercise shows how to find common values between columns.

Exercise: How do I find out if entries in one column occur in another column regardless of row position and number of times they occur?

One way to address this problem is to create two arrays for storing two columns. Then, check if the values in one array exist in the other array. Find those values that match and store them in a third array for output.

Create a Data Input node as Text File Input, and set the text file to C:\arrayTextDocument.txt in Data Jobs. Begin with the following text in the file:

Table A1.1 *c:\arrayTextDocument.txt*

A ID	B ID
0	1
1	2
3	4
5	6

6	0

Create an Expression node, and declare the following variables in the pre-expression step:

```
// Pre-Expression
// This is where we declare and
// initialize our variables.
hidden string array column_A
hidden string array column_B
hidden string array column

hidden integer column_A_size
column_A_size=1
column_A.dim(column_A_size)

hidden integer column_B_size
column_B_size=1
column_B.dim(column_B_size)

hidden integer commun_size
commun_size=1
commun.dim(commun_size)
```

All the variables are hidden and are not displayed on the output. All the arrays are defined in the beginning to be of size 1. Later, these arrays will be expanded to accommodate the number of rows that are added.

```
// Expression

// Name your First_Column field as you need
column_A.set(column_A_size, `A_ID`)

column_A_size=column_A_size+1
column_A.dim(column_A_size)

// Name the Second_Column field as you need
column_B.set(column_B_size, `B_ID`)

column_B_size=column_B_size+1
column_B.dim(column_B_size)
```

In this step, we retrieve input into the arrays and expand the size of the arrays as necessary. The size of the array might become quite large depending on the size of the column, so it is recommended you use this example with small tables.

```
// Post Expression
// This is the step where most of the
// logic will be implemented

// index to iterate through column_A
hidden integer index_column_A

// index to iterate through column_B
hidden integer index_column_B

// index to iterate through commun array
```

```
hidden integer index_commun

// index to display the commun values that were found
hidden integer commun_display_index

// string that will contain the items
// from column A when retrieving hidden string a

// string that will contain the items
// from column B when retrieving hidden string b

// String that will contain the contents of the
// commun array when retrieving
hidden string commun_content

// This boolean variable
// is to check if a commun entry has already
// been found. If so, don't display it again
hidden boolean commun_found

// This is the variable
// that will display the common entries in the end
string commun_display

// Retrieves the entries in column A
for index_column_A=1 to column_A_size Step 1
    begin
        a=column_A.get(index_column_A)
        for index_column_B=1 to column_B_size Step 1
            begin

                b=column_B.get(index_column_B)

                // Compare the entries from column A with
                // the entries from column B
                if(compare(a,b)==0)
                     begin
                // Check if this entry was already found once
                commun_found=false
                   for index_commun=1 to commun_size Step 1
                       begin
                            commun_content=commun.get(index_commun)
                            if(compare(commun_content,a)==0) then
                                 commun_found=true
                       end

                   // It is a new entry. Add it to the
                   // commun array and increment its size
                   if(commun_found==false)
                        begin
                             commun.set(commun_size,a)
                             commun_size=commun_size+1
                             commun.dim(commun_size)
                        end
                end
end
```

```
    end

// Display the contents of the commun array
// to the screen output
for commun_display_index=1 to commun_size Step 1
     begin
          pushrow()
          commun_display=commun.get(commun_display_index)
     end
```

If you want to see the output limited to the common values, add another Expression node and the following filtering code:

```
// Expression
if(isnull(`commun_display`)) then
     return false
else
     return true
```

Using Blue Fusion Functions

Once a Blue Fusion object is defined and initialized, the function methods listed can be used within the Expression node. The following exercises demonstrate how the Blue Fusion object methods can be used in the Expression node.

Exercise 1: How do I start a Blue Fusion instance and load a QKB?

To start a Blue Fusion instance and load a QKB, add the following in the **Pre-Processing** tab:

```
// Pre-processing

// defines a bluefusion object called bf
bluefusion bf;

// initializes the bluefusion object bf
bf = bluefusion_initialize()

// loads the English USA Locale
bf.loadqkb("ENUSA");
```

To load other QKBs, refer to their abbreviation. Go to the DataFlux Data Management Studio Administration riser bar and click **Quality Knowledge Base** to see which QKBs are available for your system.

Exercise 2: How do I create match codes?

To create match codes, after you initialize the Blue Fusion object with a QKB in the **Pre-Processing** tab, enter the following expressions:

```
// Expression

// define mc as the return string that contains the match code
string mc

// define the return code ret as an integer
integer ret

// define a string to hold any error message that is returned,
string error_message
// generate a match code for the string Washington D.C.,
// using the City definition at a sensitivity of 85, and
// put the result in mc
ret = bf.match code("city", 85, "Washington DC", mc);

// if an error occurs, display it; otherwise return a success message
if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Exercise 3: How do I use Blue Fusion standardize?

To use Blue Fusion standardize, enter the following expressions after you initialize the Blue Fusion object in the **Pre-Processing** tab:

```
// Expression

// define stdn as the return string that contains the standardization string stdn

// define the return code ret as an integer integer ret

// define a string to hold any error message that is returned string error_message

// standardize the phone number 9195550673,
// and put the result in stnd
ret = bf.standardize("phone", "9195550673", stdn);

//if an error occurs display it; otherwise return a success message, if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Exercise 4: How do I use Blue Fusion identify?

To use Blue Fusion identity, after you initialize the Blue Fusion object in the **Pre-Processing** tab, enter the following expressions:

```
// Expression

// define iden as the return string that contains the identification
string iden

// define the return code ret as an integer
integer ret

// define a string to hold any error message that is returned
string error_message
// generate an Ind/Org identification for IBM and put
// the result in iden
ret = bf.identify("Individual/Organization", "IBM", iden);

//if an error occurs display it; otherwise return a success message,
if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Exercise 5: How can I perform gender analysis?

To perform gender analysis, after you initialize the Blue Fusion object in the **Pre-Processing** tab, enter the following expressions:

```
// Expression

// define gend as the return string that contains the gender
string gend

// define the return code ret as an integer
integer ret

// define a string to hold any error message that is returned
string error_message
// generate a gender identification for Michael Smith,
// and put the result in gend
ret = bf.gender("name","Michael Smith",gend);

// if an error occurs display it; otherwise return a success message,
if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Exercise 6: How can I do string casing?

To perform string casing after you initialize the Blue Fusion object in the **Pre-Processing** tab, enter the following expressions:

```
// Expression

// define case as the return string that contains the case
string case

// define the return code ret as an integer integer ret

// define a string to hold any error message that is returned
string error_message

// convert the upper case NEW YORK to propercase
ret = bf.case("Proper", 3, "NEW YORK",case);

// if an error occurs display it; otherwise return a success message,
if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Exercise 7: How can I do pattern analysis?

To perform pattern analysis after you initialize the Blue Fusion object in the **Pre-Processing** tab, enter the following expressions:

```
//Expression

//define pattern as the return string
string pattern
```

```
//define the return code ret as an integer
integer ret

// define a string to hold any error message that is returned
string error_message

// analyze the pattern 919-447-3000 and output the result
// as pattern
ret = bf.pattern("character", "919-447-3000", pattern);

// if an error occurs display it; otherwise return a success message,
if ret == 0 then
  error_message = bf.getlasterror()
else
  error_message = 'Successful'
```

Using Date and Time Functions

In this section, you will find additional information about date and time functions, including:

Finding Today's Date

The following exercise describes out to find the values for today's date.

Exercise: How do I find the year, month, and day values for today's date?

To determine the parts of the current date, use the TODAY() function.

```
date today()
```

The following function returns the current date and time:

```
// Expression
date localtime
localtime=today()
```

Formatting a Date

The following exercises show now to format dates.

Exercise 1: What formats can a date have?

Dates should be in the format specified by ISO 8601 (YYYY-MM-DD hh:mm:ss) to avoid ambiguity. Remember that date types must start with and end with the # sign. For example:

Date only:

```
// Expression date dt
dt=#2007-01-10#
//Jan 10 2007
```

Date with time:

```
// Expression date dt
dt=#2007-01-10 12:27:00#
//Jan 10 2007 at 12:27:00
```

Exercise 2: How do I format the date?

To specify a format for the date in EEL, use the FORMATDATE() function:

```
string formatdate(date, string)
```

The FORMATDATE() function returns a date formatted as a string. For example:

```
// Expression
// all have the same output until formatted explicitly
date dt
dt=#2007-01-13#

string formata
string formatb
string formatc
formata=formatdate(dt, "MM/DD/YY") // outputs 01/13/07
formatb=formatdate(dt, "DD MMMM YYYY") // outputs 13 January 2007
formatc=formatdate(dt, "MMM DD YYYY") // outputs Jan 13 2007
```

Extracting Parts from a Date

To extract parts of a date, use the following exercise.

Exercise: How do I get individual components out of a date?

To extract parts of a date, use the FORMATDATE() function. For example:

```
// Expression
date dt
dt=#10 January 2003#

string year
string month
string day

// year should be 03
year=formatdate(dt, "YY")
// month should be January
month=formatdate(dt, "MMMM")
// day should be 10
day=formatdate(dt, "DD")
```

Note that if the date format is ambiguous, EEL will parse the date as MDY.

Adding or Subtracting from a Date

The following exercise explains how to add or subtract days from an existing date.

Exercise: Can I do arithmetic with dates?

EEL offers the ability to add or subtract days from an existing date. For example:

```
// Expression
date dt // variable that will contain the date
dt=#10 January 2003#
date later
date earlier

// add three days to the original date
```

```
later=dt+3
// subtract three days from the original date
earlier=dt-3
```

Comparing Dates

To compare dates, use the FORMATDATE() function.

Exercise: How do I check if two dates match and are the same?

Convert the date to a string type using FORMATDATE() function and then check for the value of the string. For example:

```
date dt

// the variable that will contain the date
// that we want to compare against
dt=#1/1/2007#

// The string variable that will contain the
// dt date in a string format
string dt_string

// The variable that will convert the
// incoming date fields to string
dt_string=formatdate(dt, "MM/DD/YY")
string Date_string

// Notice that `DATE` is the incoming field
// from the data source It is written between `` so
// it does not conflict with the date data type
Date_string=formatdate(`DATE`, "MM/DD/YY")

// boolean variable to check if the dates matched
boolean date_match

// Initialize the variable to false
date_match=false

if(compare(dt_string, Date_string)==0)then
date_match=true
```

Using Database Functions

This section explains using database functions with EEL.

Connecting to a Database

The following exercise explains how to connect to a database.

Exercise: How do I connect to a database?

To connect to a database, use the DBCONNECT() function. This function returns a dbconnection object. The syntax is:

```
dbconnection test_database
```

For example:

```
// Set connection object to desired data source
// Saved DataFlux connections can also be used
test_database=dbconnect("DSN=DataFlux Sample")
```

Listing Data Sources

Exercise 1: How do I return a list of data sources?

To return a list of data sources as a dbcursor, use the DBDATASOURCES() function.

The following example works with the Contacts table in the DataFlux sample database. Make sure you have some match codes in that table in a field called CONTACT_MATCHCODE. In the step before your expression step, use a match code generation node and have match codes created for some sample names in a text file. This text file is your job input step. Call this new field "Name_MatchCode." This example queries the Contacts table in the DataFlux sample database to see whether there are any names that match the names that you provided in your text file input.

Pre-processing window

```
// Declare Database Connection Object
dbconnection db_obj

// Declare Database Statement Object
dbstatement db_stmt
// Set connection object to desired data source
// Saved DataFlux connections can also be used
db_obj=dbconnect("DSN=DataFlux Sample")

// Prepare the SQL statement and define parameters
// to be used for the database lookup
db_stmt=db_obj.prepare("Select * from Contacts where Contact = ?")
db_stmt.setparaminfo(0,"string",30)
```

Expression window

```
// Declare Database Cursor and define fields returned from table
dbcursor db_curs
string Database_ID
string COMPANY
string CONTACT
string ADDRESS

// Set parameter values and execute the statement
db_stmt.setparameter(0,Name)
db_curs=db_stmt.select()

// Move through the result set adding rows to output
while db_curs.next()
begin
     Database_ID=db_curs.valuestring(0)
     COMPANY=db_curs.valuestring(1)
     CONTACT=db_curs.valuestring(2)
     ADDRESS=db_curs.valuestring(3)
     pushrow()
end
db_curs.release()
```

```
// Prevent the last row from occurring twice
return false
```

Using Encode and Decode Functions

Exercise 1: How do I transcode a given expression string from its native encoding into the specified encoding?

To transcode an expression, use the encode function and decode function. For example:

```
//Expression
string expression_string
expression_string="Hello World"
string decode_string
string encode_string
integer decode_return
integer encode_return
decode_return = decode("IBM1047", expression_string, decode_string)
//Decode to IBM1047 EBCDIC
encode_return = encode("IBM1047",decode_string,encode_string)
//Encode string should be "Hello World"
```

Exercise 2: What are the available encodings?

Refer to Appendix B: Encoding for a list of available encodings.

Using File Functions

File Operations

This section explains file operations in the EEL.

Exercise 1: How do I open a file?

To open a file in the EEL, use this expression:

```
// Expression
File f
open("C:\filename.txt","r")
```

The second parameter to the file object indicates the mode for opening the file (read, write, or read/write).

Exercise 2: How do I read lines from a file, treating each line as a single row from a data source?

After opening a file, use the following code to read a string line of input:

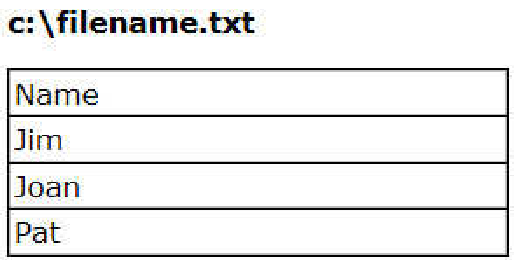
```
// Pre-Expression
File f
string input
f open("C:\filename.txt", "rw")

// Expression
input=f.readline() // Post Expression
f close()
```

Make sure that you select **Generate rows** when no parent is specified. The file cursor advances one line in the text file for each row of input from the data source.

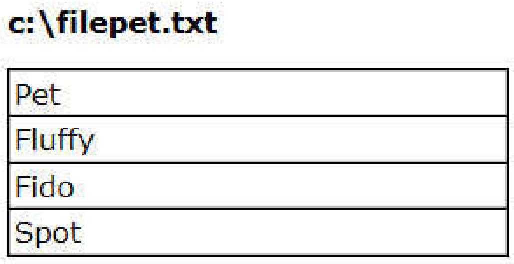
Exercise 3: How do I read lines from a text file, and create one output line for each line in the text file?

Write a WHILE loop that iterates through each line in the file with every row. For example, consider the following text files:



c:\filename.txt

Name
Jim
Joan
Pat



c:\filepet.txt

Pet
Fluffy
Fido
Spot

Use the following expression:

```
// Expression
File f
File g
string input
input='hello'
f open("C:\filename.txt")
g open("C:\filepet.txt")

while (NOT isnull(input))
     begin
input=f.readline()
print('The value of input is ' & input)
input=g.readline()
print('The value of input is ' & input)
end

seteof()

// Post Expression
f close()
```

This prints the contents of the two files to the log. If you preview the job, you see null for the input string since, the input string has a null value at the completion of the loop.

A good way to see how this example works in your job is to add an expression step that sets the end of file:

```
// Expression
seteof()
```

The preview pane shows the value of input as null, but the log pane shows each of the possible values listed in the filename.txt and filepet.txt files.

Exercise 4: How do I write to a file?

To write to a file, use the WRITELINE() function in the file object. For example:

```
// Expression File f
f open("C:\filename.txt", "w")
f writeline("Hello World ")

// Post Expression
f close()
```

Note: This function overwrites the current contents of your text file.

Exercise 5: How do I move from one position to another in a file?

To move from one position in a file to another, there are three available functions: SEEKBEGIN(), SEEKCURRENT(), and SEEKEND().

The SEEKBEGIN() function sets the file pointer to a position starting at the beginning of the file. It returns true on success. Otherwise, it returns false. The parameter specifies the position:

```
seekbegin([position])
```

The SEEKCURRENT() function sets the file pointer to a position starting at the current position. It returns true on success. Otherwise, it returns false. The parameter specifies the number of bytes from the current position:

```
seekcurrent([position])
```

The SEEKEND() function sets the file pointer to a position starting at the end of the file. It returns true on success. Otherwise, it returns false. The parameter specifies the position from the end of the file:

```
seekend([position])
```

All of these functions receive as a parameter the number of bytes to move from the current position in the file. Specify 0 in the SEEKBEGIN() or the SEEKEND() functions to go directly to the beginning or the end of the file. For example: In order to append to the end of a file that you would select **Generate rows** when no parent is specified, and enter:

```
// Expression File f
f open("C:\Text_File\file_content.txt", "rw")
f seekend(0)
f writeline("This is the end ")
seteof()
```

This example adds the text "This is the end" to the end of the file. If you move to the beginning of the file, use the WRITELINE() function to overwrite existing content.

Close the file with F.CLOSE() in the post-processing step:

```
// Post Processing
f close()
```

Exercise 6: How do I copy the contents of a file to another file?

To copy the contents of one file to another, use the Boolean function, COPYFILE(). This function takes the originating filename as the first parameter and the destination filename as the second parameter. The destination file can be created or amended by this function. For example:

```
// Expression
string names
string pets
names="C:\filename.txt" pets="C:\filecopy.txt"

copyfile(names, pets)

seteof()
```

Exercise 7: How do I read or write a certain number of bytes from a text file?

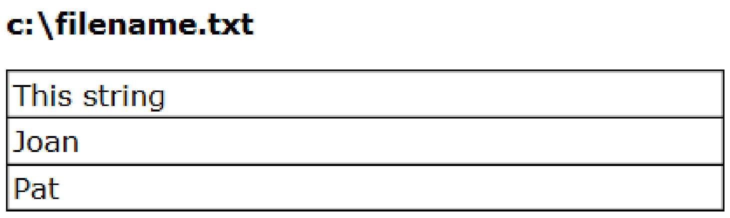
To read a specified number of bytes from a text file, use the READBYTES() function:

```
string input File a
a.open("C:\filename.txt", "r")
a.readbytes(10, input)
```

To write a specified number of bytes to a text file, use the WRITEBYTES() function:

```
string input
input="This string is longer than it needs to be." File b
b.open("C:\filename.txt", "rw")
b.writebytes(10, input)
```

By overwriting existing data, this expression produces the following:



c:\filename.txt

This string
Joan
Pat

Manipulating Files

Exercise 1: How do I retrieve information about the file?

To determine whether a file exists, use the FILEEXISTS() function:

```
boolean fileexists(string)
```

The FILEEXISTS() function returns true if the specified file exists. The string parameter is the path to the file.

To find the dates a file was created and modified, use the FILEDATE() function:

```
date filedate (string, boolean)
```

The FILEDATE() function returns the date on which a file was created. If the second parameter is true, it returns the modified date.

For example:

```
// Expression
boolean file_test
date created
date modified

file_test=fileexists("C:\filename.txt")
created=filedate("C:\filename.txt", false)
```

```
modified=filedate("C:\filename.txt", true)

seteof()
```

Note: If the FILEDATE() function returns a null value but the FILEEXISTS() function returns true, you most likely entered the file path incorrectly.

To get the size of a file, you can use the OPEN(), SEEKEND(), and POSITION() functions. The size of the file is returned in bytes. For example:

```
// Expression
File f
integer byte_size

f.open("C:\filename.txt", "rw")
f.seekend(0)

// The integer variable byte_size will have
// the size of the file in bytes
byte_size=f.position()
```

Exercise 2: Is it possible to perform operations such as renaming, copying, or deleting a file?

Yes. To delete a file, use the DELETEFILE() function:

```
boolean deletefile(string)
```

This action deletes a file from the disk. The string parameter is the path to the file.

Note: Use care when using this function. Once you delete a file, it is gone.

To move or rename a file, use the MOVEFILE() function:

```
boolean movefile(string, string)
```

For example, the following code moves filename.txt from the root to the Names folder.

```
boolean newLocation
newLocation=movefile("C:\filename.txt","C:\Names\filename.txt")
seteof()
```

Note: The directory structure must already be in place for the function to move the file to its new location.

Using Integer and Real Functions

Integers and real types are basic data types in EEL. An integer can be converted to a real type, and a real type value can be converted to an integer. This section focuses on available functions in EEL that work on integers and real types.

Determining Type

Determine the type of a variable before doing calculations.

Exercise: How do I determine whether the variable has a numeric value?

The ISNUMBER() built-in function can be used to determine whether a variable has a numeric value. It takes a variable as a parameter and returns true if the expression is a number. For example:

```
// Expression
string str
string input
input=8 // although a string, input is coerced into an integer

if(isnumber(`Input`))
     str="this is a number" // input is a number
else
     str="this is a string"
```

Assigning Values

The following exercise provides information about assigning values.

Exercise: Can integers and real types have negative values?

Yes, integers and real types are not limited to positive values. Add a negative sign in front of the value to make it negative. For example:

```
// Expression
integer positive
integer negative
positive=1
negative=-1 // negative is equal to -1
```

Casting

The need to coerce from one type to another can happen frequently with integers, real data types, and string types. The user does not have to perform any task; EEL handles the casting automatically.

Exercise 1: Can I assign the value of a real data type to an integer? What about assigning an integer to a real data type?

Yes, integers and real types can be changed from one type to the other. To change the type, assign one type to the other.

Exercise 2: Is it possible to combine integers and real data types with strings?

Yes, a string type can be changed to an integer or a real type. For example:

```
integer x

// string is converted to value 10
// x will have the value 15
x=5 + "10"
```

Exercise 3: Is it possible to assign the integer value zero to a Boolean to represent false?

In EEL, Boolean values can have an integer value of zero, which is interpreted as false. Any nonzero integer value is interpreted as true.

Range and Precision

When working with scientific data with either very small or very large values, the range and precision of the integer and real types might be important.

Exercise: What is the range or precision for real and integer values?

Integer types are stored as 64-bit signed quantities with a range of –2 * 10^63 to 2

Real types are stored as high precision values with an approximate precision of 44 digits and a range of 5.0 * 10^-324 to 1.7 * 10^308. Real types are based on the IEEE 754 definition.

List of Operations

In EEL, the following operations can be performed on real and integer types.

Exercise: What operations can I do on real and integer types?

The list of operations for real and integer types includes:

- Multiplication (*)
- Division (/)
- Modulo (%)
- Addition (+)
- Subtraction (–)

Currently, it is not possible to perform trigonometric or logarithmic calculations. You can perform exponential calculations using the POW() function:

```
real pow(real,real)
```

The POW() function returns a number raised to the power of another number.

```
// Expression
real exponential

// exponential is 8
exponential=pow(2,3)
```

Rounding

Integers and real values in EEL can be rounded using the round() function. The second parameter is an integer value that determines how many significant digits to use for the output. A positive value is used to round to the right of the decimal point. A negative value is used to the left of the decimal point.

Exercise: Can integer and real types be rounded?

Yes, by using the ROUND() function. Consider the following code example:

```
// Expressions
integer integer_value
integer_value=1274
real real_value
real_value=10.126

integer ten
integer hundred
integer thousand

// the value for ten will be 1270
ten=round(integer_value,-1)

// the value for hundred will be 1300
```

```
hundred=round(integer_value,-2)

// the value for thousand will be 1000
thousand=round(integer_value,-3)

real real_ten real
real_hundred

// the value for real_ten will be 10.1
real_ten= round(real_value, 1)

// the value for real_hundred will be 10.13
real_hundred=round(real_value, 2)
```

Using Regular Expression Functions

Using Regular Expressions

For a regular expression (regex) to work, you must first compile. In the Data Job Editor, this is best done in the pre-processing step. Here are some examples.

Exercise 1: How do I find matches within a string?

To find the first match in the string, use the FINDFIRST() function. To find subsequent matches in the string, use FINDNEXT(). For example:

```
regex r
r.compile("a.c")
if r.findfirst("abcdef")
     print("Found match starting at " & r.matchstart() & " length " &
r.matchlength())
```

Exercise 2: How do I know whether my regex pattern matches part of my input?

To see whether your regex pattern finds a match in the input string, follow this example:

```
regex a
boolean myresult

a.compile("a","ISO-8859-7")
myresult=a.findfirst("abc")
```

Exercise 3: How do I find the regex pattern I want to match?

To find the first instance of the regex pattern that you want to match, follow this example:

```
integer startingPosition regex r
r.compile("a.c")
if r.findfirst("abcdef")
     startingPosition=r.matchstart()
```

Exercise 4: How do I replace a string within my regex?

To replace a string, compile the regex and use the replace function as follows:

```
regex r
r.compile("xyz")
```

```
r.replace("abc","def")
```

This exercise replaces "abc" with "def" within the compiled "xyz."

Using String Functions

Determining Type

The following exercises demonstrate how to determine the data type of a string.

Exercise 1: How do I determine whether an entry is a string?

To determine whether the string is a string type, use the TYPEOF() function:

```
string typeof(any)
```

The TYPEOF() function returns the type of data that the expression converts to. For example:

```
// Expression
string hello
hello="hello"

boolean error
error=false

// variable that will contain the type
string type
type=typeof(hello)

// type should be string
if(type<>"string") then
    error=true
```

Exercise 2: How do I determine whether a string consists of alphabetic characters?

To determine whether a string is made up entirely of alphabetic character, use the ISALPHA() function:

```
boolean isalpha(any)
```

The ISALPHA() function returns a value of true if the string is made up entirely of alphabetic characters. For example:

```
// Expression
string letters
letters="lmnop"
string mixed
mixed="1a2b3c"

string alphatype
alphatype=isalpha(letters) // true
string mixedtype
mixedtype=isalpha(mixed) // false
```

Exercise 3: How can I retrieve all values that are either not equal to X or null values?

To retrieve the above stated values, use the ISNULL() function:

```
boolean isnull(any)
```

For example:

```
// Expression
if State <> "NC" OR isnull(State)
     return true
else
     return false
```

Extracting Substrings

Exercise: How do I get substrings from an existing string?

To get substrings, there are three available functions: LEFT(), RIGHT(), and MID().

To return the leftmost characters of a string, use the LEFT() function:

```
string left(string, integer)
```

To return the rightmost characters of a string, use the RIGHT() function:

```
string right(string, integer)
```

For example:

```
// Expression
string greeting
greeting="Hello Josh and John"

string hello
string John
string inbetween

hello=left(greeting,5) // "Hello"
John=right(greeting,4) // "John"
inbetween=left(greeting, 10) // "Hello Josh"
inbetween=right(inbetween, 4) // "Josh"
```

Another approach is to use the MID() function:

```
string mid(string, integer p, integer n)
```

The MID() function returns a substring starting at position p for n characters. For example:

```
string substring
// substring will be the string "Josh"
substring=mid(greeting, 7, 4);
```

Parsing

Exercise: How do I parse an existing string into smaller strings?

To parse a sting, use the APARSE() function:

```
integer aparse(string, string, array)
```

The APARSE() function parses a string into a string array. The number of elements in the array is returned. For example:

```
// Expression
```

```
string dataflux
dataflux="Dataflux:dfPower:Architect"

// An array type to contain the parsed words
string array words

// integer to count the number of words
integer count

// count will have a value of 3
count=aparse(dataflux, ":", words)

string first_word
first_word=words.get(1) // First word will be "DataFlux"

string second_word
second_word=words.get(2) // Second word will be "Data Management"

string third_word
third_word=words.get(3) // Third Word will be "Studio"

string last_entry // This will have the last entry.
last_entry=words.get(count)
```

The APARSE() function is useful if you want to retrieve the last entry after a given separator.

Similar to the APARSE() function is the PARSE() function. The syntax for the PARSE() function is:

```
integer parse(string, string, ...)
```

The PARSE() function parses a string using another string as a delimiter. The results are stored starting from the third parameter. It returns the total number of parameters.

You would use the PARSE() function in the following situation:

```
// Expression
integer count

string first
string second
string third

// first contains "DataFlux"
// second contains "Data Management"
// third contains "Studio"
count=parse("DataFlux:Data Management:Studio", ":", first, second,
third);
```

The main difference between the two functions is that APARSE() is suitable for arrays while PARSE() is useful for returning individual strings.

Converting ASCII Characters

EEL has the ability to convert characters to their ASCII values, and to convert ASCII values to characters.

Exercise: Is it possible to convert between ASCII characters and values?

Yes. To convert between ASCII characters and values, use the CHR() and ASC() functions. For example:

```
// Expression
integer ascii_value
string character_content

ascii_value=asc("a"); // ascii_value is 97
character_content=chr(97) // returns the letter "a"
```

For a complete list of ASCII values, see Appendix A: ASCII Printable Characters.

Manipulating Strings

Frequently, when working with strings, you might want to perform manipulations such as adjusting the case, removing spaces, concatenating strings, and getting the length of a string. EEL has built-in functions to perform these actions.

Exercise 1: How do I concatenate strings?

To concatenate a string, use the "&" symbol. For example:

```
// Expression
string Hello
Hello="Hello "

string World
World=" World"

string Hello_World
Hello_World=Hello & World // outputs "Hello World"
```

Exercise 2: How do I get the length of a string and remove spaces?

To get the length of a string, use the LEN() function, and then to remove the spaces, use the TRIM() function.

The LEN() function returns the length of a string:

```
integer len(string)
```

The TRIM() function returns the string with the leading and trailing whitespace removed:

```
string trim(string)
```

For example:

```
// Expression
string content
content="       spaces          "

integer content_length

content=trim(content) // Remove spaces

// returns 6
content_length=len(content)
```

Exercise 3: How do I convert a string type to lowercase or uppercase?

To convert a string to lowercase or uppercase, use the LOWER() and UPPER() functions.

The LOWER() function returns the string in lowercase:

```
string lower(string)
```

The UPPER() function returns the string in uppercase:

```
string upper(string)
```

This function returns the edit distance between two strings. Specifically, this function returns the number of corrections that would need to be applied to turn one string into the other.

The following examples use these functions:

```
// Expression
integer difference
integer comparison

string hello
hello="hello"

string hey
hey="hey"

// comparison is -1 because hello comes before
hey comparison = compare(hello, hey, true);

// difference is 3 because there are three different letters
difference = edit_distance(hello, hey);
```

Exercise 2: How do I check if a string matches, or if it is a substring inside another string?

The following built-in EEL functions handle this situation.

The INSTR() function returns the location of one string within another string, stating the occurrence of the string.

```
boolean instr(string, string, integer)
```

The MATCH_STRING() function determines whether the first string matches the second string, which might contain wildcards.

```
boolean match_string(string, string)
```

Search strings can include wildcards in the leading (*ABC) and trailing (ABC*) position, or a combination of the two (*ABC*). Wildcards within a string are invalid (A*BC). Question marks can be used as a wildcard, but can be matched only to a character. For example, AB? matches ABC, not AB. To execute a search for a character that is used as a wildcard, precede the character with a backslash. This denotes that the character should be used literally and not as a wildcard. Valid search strings include: *BCD*, *B?D*, *BCDE, *BC?E, *BCD?, ABCD*, AB?D*, ?BCD*, *B??*, *B\?\\* (will match the literal string AB?\E). An invalid example is: AB*DE. For more complex searches, use regular expressions instead of the MATCH_STRING() function.

Consider the following code example with these functions:

```
// Expression
```

```
string content
content="Monday is sunny, Tuesday is rainy & Wednesday is windy"

string search
search="*Wednesday is windy" // note the * wildcard

integer found_first
integer found_next

boolean match

// Check if the search string is in the content match=match_string(content, search)

if (match) then
     begin
          // Will find the first occurrence of day
          found_first=instr(content, "day", 1)

          // Will find the second occurrence of day
          found_next=instr(content, "day", 2)
       end
```

Exercise 3: How do I know when the correct Surviving Record is selected as a survivor?

When comparing integers in the EEL, it is important to use the correct variable type. When using the variable type "string", a string value of "5" is actually greater than the string value of "10". If the values need to be compared as integers, the values must be converted into variables of integer type first. To accomplish this conversion use the TOINTEGER() function.

Replacing Strings

The REPLACE() function replaces the first occurrence of one string with another string and returns the string with the replacement made.

```
string replace(string, string, string, integer)
```

If the fourth parameter is omitted or set to 0, all occurrences are replaced in the string. If the fourth parameter is set to another number, that many replacements are made.

Consider the following example:

```
// Expression
string starter
string replace
string replaceWith
string final

starter="It's a first! This is the first time I came in first place!"
replace="first"
replaceWith="second"

final =replace(starter, replace, replaceWith, 2)

seteof()
```

This example produces the following results:

starter	replace	replaceWith	final
It’s a first! This is the first time I came in first place!	first	second	It's a second! This is the second time I came in first place!

Finding Patterns

It is possible to extract patterns out of strings using EEL. EEL identifies the following as part of a string's pattern: 9 = numbers a = lowercase letters A = uppercase letters

Exercise: How do I get a string pattern?

To determine the string pattern, use the PATTERN() function:

```
string pattern(string)
```

The PATTERN() function indicates whether a string has numbers or uppercase and lowercase characters. It generates a pattern from the input string. For example:

```
// Expression
string result;
string pattern_string;
pattern_string="abcdeABCDE98765";

// The result will be aaaaaAAAAA99999
result=pattern(pattern_string);
```

Identifying Control Characters

EEL can identify control characters such as a horizontal tab and line feed.

Exercise: How can I detect control characters in a string?

To detect control characters, use the HAS_CONTROL_CHARS() function.

```
boolean has_control_chars(string)
```

The HAS_CONTROL_CHARS() function determines whether the string contains control characters. For a list of control characters, see Appendix A: ASCII Control Characters.

Evaluating Strings

EEL enables you to dynamically select the value of a field.

Exercise: How can I convert field names into values?

To convert field names into values, use the VAREVAL() function.

```
string vareval(string)
```

The VAREVAL() function evaluates a string as if it were a variable.

Note: Since it has to look up the field name each time it is called, VAREVAL() is a slow function and should be used sparingly.

In the following example, you have incoming data from three fields: field1, field2, and field3, as shown in the following table.

Table A1.2 *C:\varevalExample.txt*

field_1	**field_2**	**field_3**	**field_4**	**field_5**
1	Bob Brauer	123 Main St.	Cary	NC
2	Don Williams	4 Clover Ave.	Raleigh	NC
3	Mr. Jim Smith	44 E. Market Street	Wilmington	NC
4	Ms. Amber Jones	300 Chatham Dr.	Durham	NC
5	I Alden	99 A Dogwood Ave.	Apex	NC

You can write a for loop that builds the string ("field" and n), and uses the VAREVAL() function to get the value of the field. For example:

```
// Pre-expression
string field_number
string field_value

// Expression
hidden integer n
for n=1 to 5
begin
         field_number='field_' & n
         field_value=vareval(field_number)
         n=n+1
         pushrow()
end

// this next statement prevents the last row from showing up twice
return false
```

Appendix 2
Reserved Words

The following list of reserved words cannot be used for label names:

and	date	hidden	pointer	step
array	else	if	private	string
begin	end	integer	public	then
Boolean	for	not	real	to
bytes	global	null	return	visible
call	goto	or	static	while

Appendix 3
ASCII Values

ASCII Values

The following table contains the ASCII printable characters that can be represented by decimal values:

Table A3.1 *ASCII Printable Characters*

Value	Character	Value	Character	Value	Character
32	(space)	64	@	96	‘
33	!	65	A	97	a
34	“	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	‘	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m

Value	Character	Value	Character	Value	Character
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
62	?	95	_		

ASCII Control Characters

The following table contains the ASCII control characters that can be represented by decimal values:

Value	Character	Value	Character	Value	Character
0	Null character	11	Vertical tab	22	Synchronous idle
1	Start of header	12	Form feed	23	End of transmission block

Value	Character	Value	Character	Value	Character
2	Start of text	13	Carriage return	24	Cancel
3	End of text	14	Shift out	25	End of medium
4	End of transmission	15	Shift in	26	Substitute
5	Enquiry	16	Data link escape	27	Escape
6	Acknowledgment	17	Device control 1	28	File separator
7	Bell	18	Device control 2	29	Group separator
8	Backspace	19	Device control 3	30	Record separator
9	Horizontal tab	20	Device control 4	31	Unit separator
10	Line feed	21	Negative acknowledgment	127	Delete

Appendix 4

Encoding

The table below explains the options available with the **Encoding** drop-down list. In most cases, you will select **Default** from the **Encoding** drop-down list.

Option	Character Set	Encoding Constant	Description
hp-roman8	Latin	19	An 8-bit Latin character set.
IBM437	Latin	32	Original character set of the IBM PC. Also known as CP437.
IBM850	Western Europe	33	A code page used in Western Europe. Also referred to as MS-DOS Code Page 850.
IBM1047	EBCDIC Latin 1	10	A code page used for Latin 1.
ISO-8859-1	Latin 1	1	A standard Latin alphabet character set.
ISO-8859-2	Latin 2	2	An 8-bit character sets for Western alphabetic languages such as Latin, Cyrillic, Arabic, Hebrew, and Greek. Commonly referred to as Latin 2.
ISO-8859-3	Latin 3	13	An 8-bit character encoding. Formerly used to cover Turkish, Maltese, and Esperanto. Also known as "South European".

Option	Character Set	Encoding Constant	Description
ISO-8859-4	Latin 4	14	An 8-bit character encoding originally used for Estonian, Latvian, Lithuanian, Greenlandic, and Sami. Also known as "North European".
ISO-8859-5	Latin/Cyrillic	3	Cyrillic is an 8-bit character set that can be used for Bulgarian, Belarusian, and Russian.
ISO-8859-6	Latin/Arabic	9	This is an 8-bit Arabic (limited) character set.
ISO-8859-7	Latin/Greek	4	This is an 8-bit Arabic (limited) character set.
ISO-8859-8	Latin/Hebrew	11	Contains all of the Hebrew letter without Hebrew vowel signs. Commonly known as MIME.
ISO-8859-9	Turkish	5	This 8-bit character set covers Turkic and Icelandic. Also known as Latin-5.
ISO-8859-10	Nordic	15	An 8-bit character set designed for Nordic languages. Also known as Latin-6.
ISO-8859-11	Latin/Thai	6	An 8-bit character set covering Thai. Might also use TIS-620.
ISO-8859-13	Baltic	16	An 8-bit character set covering Baltic languages. Also known as Latin-7 or "Baltic Rim".
ISO-8859-14	Celtic	17	An 8-bit character set covering Celtic languages like Gaelic, Welsh, and Breton. Known as Latin-8 or Celtic.

Option	Character Set	Encoding Constant	Description
ISO-8859-15	Latin 9	18	An 8-bit character set for English, French, German, Spanish, and Portuguese, as well as other Western European languages.
KOI8-R	Russian	12	An 8-bit character set covering Russian.
Shift-JIS	Japanese		Based on character sets for single-byte and double-byte characters. Also known as JIS X 0208.
TIS-620	Thai	20	A character set used for the Thai language.
UCS-2BE	Big Endian	7	Means that the highest order byte is stored at the highest address. This is similar to UTF-16.
UCS-2LE	Little Endian	8	Means the lowest order byte of a number is stored in memory at the lowest address. This is similar to UTF-16.
US-ASCII	ASCII	31	ASCII (American Standard Code for Information Interchange) is a character set based on the English alphabet.
UTF-8	Unicode		An 8-bit variable length character set for Unicode.
Windows-874	Windows Thai	21	Microsoft Windows Thai code page character set.

Option	Character Set	Encoding Constant	Description
Windows-1250	Windows Latin 2	22	Windows code page representing Central European languages like Polish, Czech, Slovak, Hungarian, Slovene, Croatian, Romanian, and Albanian. This option can also be used for German.
Windows-1251		23	
Windows-1252	Windows Latin 1	24	Nearly identical with Windows-1250.
Windows-1253	Windows Greek	25	A Windows code page used for modern Greek.
Windows-1254	Windows Turkish	26	Represents the Turkish Windows code page.
Windows-1255	Windows Hebrew	27	This code page is used to write Hebrew.
Windows-1256	Windows Arabic	28	This Windows code page is used to write Arabic in Microsoft Windows.
Windows-1257	Windows Baltic	29	Used to write Estonian, Latvian, and Lithuanian languages in Microsoft Windows.
Windows-1258	Windows Vietnamese	30	This code page is used to write Vietnamese text.

Appendix 5 Legal Notices

Apache Portable Runtime License Disclosure

Apache/Xerces Copyright Disclosure

Boost Software License Disclosure

Canada Post Copyright Disclosure

DataDirect Copyright Disclosure

Expat Copyright Disclosure

gSOAP Copyright Disclosure

IBM Copyright Disclosure

Informatica Address Doctor Copyright Disclosure

Loqate Copyright Disclosure

Microsoft Copyright Disclosure

Oracle Copyright Disclosure

PCRE Copyright Disclosure

A modified version of the open source software PCRE library package, written by Philip Hazel and copyrighted by the University of Cambridge, England, has been used by DataFlux for regular expression support. More information on this library can be found at: ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/.

Red Hat Copyright Disclosure

SAS Copyright Disclosure

SQLite Copyright Disclosure

Sun Microsystems Copyright Disclosure

TomTom Copyright Disclosure

USPS Copyright Disclosure

VMware Copyright Disclosure