DataFlux Expression Language Reference Guide

DATAFLUX
A sas COMPANY

YOUR DATA.
YOUR BUSINESS.
ONE SOLUTION.

This page is intentionally blank

DataFlux Expression Language Reference Guide for Data Management Studio

Version 2.1.1

September 1, 2010

This page is intentionally blank

Contact DataFlux

Corporate Headquarters

DataFlux Corporation

940 NW Cary Parkway, Suite 201
Cary, NC 27513-2792
Toll Free Phone: 877-846-FLUX (3589)
Toll Free Fax: 877-769-FLUX (3589)
Local Phone: 1-919-447-3000
Local Fax: 919-447-3100
Web: http://www.dataflux.com

DataFlux United Kingdom

Enterprise House
1-2 Hatfields
London
SE1 9PG
Phone: +44 (0) 20 3176 0025

DataFlux Germany

In der Neckarhelle 162
69118 Heidelberg
Germany
Phone: +49 (0) 6221 4150

DataFlux France

Immeuble Danica B
21, avenue Georges Pompidou
Lyon Cedex 03
69486 Lyon
France
Phone: +33 (0) 4 72 91 31 42

Technical Support

Phone: 1-919-531-9000
Email: techsupport@dataflux.com
Web: http://www.dataflux.com/MyDataFlux-Portal

Documentation Support

Email: docs@dataflux.com

Legal Information

DataFlux Legal Statements

DataFlux Solutions and Accelerators Legal Statements

DataFlux Legal Statements

Apache Portable Runtime License Disclosure

Apache/Xerces Copyright Disclosure

DataDirect Copyright Disclosure

Expat Copyright Disclosure

gSOAP Copyright Disclosure

IBM Copyright Disclosure

Microsoft Copyright Disclosure

Oracle Copyright Disclosure

PCRE Copyright Disclosure

Red Hat Copyright Disclosure

SAS Copyright Disclosure

Java Toolkit

Table of Contents

Introduction....1

Conventions Used In This Document 1

DataFlux Reference Publications 1

Expression Engine Language....2

Overview of the Expression Engine Language 2

Declaration of Symbols 4

Statements 6

Expressions....10

Functions 15

Objects....16

Expression Engine Language Functions....17

Array Functions....18

Blue Fusion Functions 21

Boolean Functions 32

Database Functions 32

Data Input Functions 37

Date and Time Functions....44

Event Functions 47

Execution Functions....48

External File Functions 50

Information/Conversion Functions 66

Logging Functions 73

Macro Variable Functions....76

Mathematical Functions....80

Regular Expression Functions....86

Search Functions....................99
String Functions....................100
Experimental Functions....................125
Data Job Expressions Node 133
Technical Support 134
Frequently Asked Questions....................134
Appendixes 180
Appendix A: Reserved Words 180
Appendix B: ASCII Values....................181
Appendix C: Encoding 182
Glossary 185
Index 186

Introduction

This section provides basic information about the product and documentation.

Conventions Used In This Document

This document uses several conventions for special terms and actions.

Typographical Conventions

The following typographical conventions are used in this document:

Bold	Text in bold signifies a button or action
italic	Identifies document and topic titles
`monospace`	Typeface used to indicate examples of code

Syntax Conventions

The following syntax conventions are used in this document:

[]	Brackets [] are used to indicate variable text, such as version numbers
#	The pound # sign at the beginning of example code indicates a comment that is not part of the code
//	Two slashes at the beginning of example code indicates a comment that is not a part of the code
>	The greater than symbol is used to show a browse path, for example **Start** > **Programs** > **DataFlux** > **DataFlux Data Management Studio [version]**
italic	Identifies a function, method, or object

DataFlux Reference Publications

This document may reference other DataFlux® publications including:

DataFlux Data Management Studio Installation and Configuration Guide

DataFlux Data Management Studio Online Help

DataFlux Quality Knowledge Base Online Help

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 (Studio). The Expression Engine Language (EEL) provides many statements, functions, and variables for manipulating data.

> **Caution:** It is recommended that you have some programming experience before using the EEL.

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 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 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 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 on using macro variables. The results from the Expression node are determined by the code in the Expression Properties dialog.

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

Overview of the Expression Engine Language

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. Declarations must be located in the code outside of programmatic constructs, so declaring a variable in a *for* loop is illegal.

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 new-line characters for readability. Code may include comments. Comments are text within a code segment that are 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 there are two symbols (output fields from the previous step) named "x" and "y." Following 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"|"pointer"
```

and identifier is a non-keyword word starting with an alphabetic character followed by characters, digits, underscores, or any string delimited by back quotes (`). 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 only visible 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 may 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 2GB. However this only applies to fields within the Expression node. If the symbol is available in the output it is truncated to 32k when the Expression node passes the value on to the next node. Therefore, if for example you define a string of length 45k, you can work with it inside the expression node, but on output it is truncated to 32k.

- The keyword, *bytes*, qualifies a string size in bytes. See previous note for additional details.
- Symbols may 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 Data Management Studio - Data Job Editor, 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 will 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.
- To declare a variable with spaces or other special characters in the name, write your variable name between back quotes (`). For example:

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

  Note: It is the grave accent character (`), also known as the back quote, that is employed, 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`
```

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 only available to Post-Processing.
- Automatic symbols, which are symbols from the previous step, are available to any of the three blocks.

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 may optionally be separated by a semicolon or new-line character. To group more than one statement together (for example, in a *for* loop), use *begin/end*.

Goto and Label

Syntax: goto label

Syntax: 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 may 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 string symbol.

For example:

```
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 on arrays, see Arrays.

Return

Syntax: "return" expression

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

- In the Data Jobs, return type is converted to boolean.
- If a false value is returned from 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

Syntax: "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.
- The expressions are only evaluated before the loop begins.
- If the step value is not specified, it defaults to one.
- 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

Syntax: "while" expression statement

The *while* loop allows you to execute the same code multiple times while 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

Syntax: "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 may be nested as well.

Call

Syntax: ["call"] function

This statement calls a function and discards the return value.

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, boolean, and pointer.
- 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
(,)	parentheses (can be nested to any depth)
* / %	multiply divide modulo
+ -	add subtract

Operators	Description
&	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:

$$a = qd + r \text{ and } 0 \leq r < | d |, \text{ where } | d | \text{ denotes the absolute value of } d$$

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

The comparison operator (==) should not be confused 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 infomration 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 will be evaluated and the following values will be considered true, (upper/lower/mixed): true, t, yes, y, or 1. The following values will be considered false (also upper/lower/mixed): false, no, n, or 0.

For more 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 Jan 1, 1900, and the fraction representing the fraction of a day.
- 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.

For example:

```
date dt
// Jan 10 2003
dt=#01/10/03#
dt=#10 January 2003#
dt=#Dec 12 2001 11:59:20#
// date is now Dec 15
dt=dt + 3

// prints "15 December '01"
call print(formatdate(dt,"DD MMMM 'YY"))
// sets dt to the current date and time
dt=today()
```

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 Propagation

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	runtime error if any of the terms are null
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 some 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	To String	To Integer	To Real	To Date	To Boolean	To Pointer
from String		yes	yes	yes	yes	no
from Integer	yes		yes	yes	yes	no
from Real	yes	yes		yes	yes	no
from Date	yes	yes	yes		no	no
from Boolean	yes	yes	yes	no		no
from Pointer	no	no	no	no	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.
string to date	Most date formats are recognized and intelligently converted. Years between 51 and 99 are assumed to be in 1900, others are in 2000.
string to boolean	The values yes, no, true, false, y, n, t, and f are recognized.
integer to real to boolean	Any non-zero value is true. Zero is false.

Functions

- A function may 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 of 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 may 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)
```

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:

- Blue Fusion - Expressions and Functions
- Databases - Database connectivity (dbconnect object)
- Files - Text file reading and writing (file object)
- Regular Expressions - Regular expression searches (regex object)

Expression Engine Language Functions

The following table lists Expression Engine Language (EEL) functions, syntax, and descriptions. Click a function category for additional information.

- Array Functions
- Blue Fusion Functions
- Boolean Functions
- Database Functions
- Data Input Functions
- Date and Time Functions
- Event Functions
- Execution Functions
- File Functions
- Information/Conversion Functions
- Logging Functions
- Macro Variable Functions
- Mathematical Functions
- Regular Expression Functions
- Search Functions
- String Functions

  Note: There is a new keyword, bytes, for string size. The 255 in the declaration string(255) refers to characters, while in string(255 bytes) the number refers to 255 bytes. Specifically, string(255 bytes) is used for multi-byte languages. For more information, see Declaration of Symbols.

- Experimental 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: *set*, *get*, and *dim*.

- dim
- get
- set

dim Function

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

Category: Array

Syntax

arrayName.*dim*[(newsize)]

Arguments

arrayName

> is the name of the array that you have declared earlier in the process

newsize

> [optional] is the numeric size (dimension) of the array, this can be specified as a numeric constant, field name, or expression

returns

> an integer representing the current size of the array

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 re-sized. The supported array types include:

- String
- Integer
- Date
- Boolean
- Real

Examples

The following statements illustrate the *dim* function:

Example #1

```
// 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
rc = 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

Syntax

<array name>.*get*(n)

Arguments

array name

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

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

returns

a string with the content of the specified array element that was retrieved

Details

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

Examples

The following statements illustrate the *get* function:

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 value of the array.

Category: Array

Syntax

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

Arguments

array name

is the name of the array that you have declared earlier in the process n

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

string

    is the value you wish to place into the array element, this can be specified as a string constant, field name, or expression

returns

    boolean [true = success; false = error]

Details

The *set* function sets the value of an entry in the array.

Examples

The following statements illustrate the *set* function:

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")
```

Blue Fusion Functions

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

The Blue Fusion functions supported within the Expression node are:

- bluefusion.case
- bluefusion.gender
- bluefusion.getlasterror
- bluefusion.identify

- bluefusion_initialize
- bluefusion.loadqkb
- bluefusion.matchcode
- bluefusion.pattern
- bluefusion.standardize

bluefusion.case Function

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

Category: Blue Fusion

Syntax

bluefusion.case (<case_def>, casing_type, input, result)

Arguments

case_def

    [optional] a string representing the name of a case definition in the QKB

casing_type

    integer numeric constant that specifies the type of casing that is applied, [1 = upper case, 2 = lower case, 3 = proper case]

input

    a string representing the input value or input field name

result

    a string representing the output field name

returns

    Boolean [1 = success, 0 = error]

Details

The *bluefusion.case* function applies casing rules to an input string and outputs the result to a field.

The function is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized using a call to the function *bluefusion_initialize*.

You may specify one of three casing types: upper, lower, or proper case. When uppercase or lowercase is specified, the function applies Unicode uppercase or lower case 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 may optionally 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 may 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 wish 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 *bluefusion.loadqkb* before calling *bluefusion.case*. The function *bluefusion.loadqkb* loads the contents of a QKB into memory and links that QKB with the *bluefusion* object. This enables *bluefusion.case* to access the case definition you specify.

Examples

The following statements illustrate the *bluefusion.case* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.case("", 1, "ronald mcdonald", output)
 // outputs "RONALD MCDONALD"

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

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

bluefusion.gender Function

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

Category: Blue Fusion

Syntax

bluefusion.gender(gender_def, input, result)

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

returns

    Boolean [1 = success, 0 = error]

Details

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

The function is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized using a call to the function *bluefusion_initialize*. The member function *bluefusion.loadqkb* must then be called to load the contents of a QKB into memory and link that QKB with the *bluefusion* object. The *bluefusion* object then retains information about the QKB and the QKB locale setting.

When calling *bluefusion.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.

Examples

The following statements illustrate the *bluefusion.gender* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.loadqkb("ENUSA")
 bf.gender("Name", "John Smith", output)
 // outputs "M"

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

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

bluefusion.getlasterror Function

Returns a string describing the most recent error encountered by a *bluefusion* object.

Category: Blue Fusion

Syntax

bluefusion.getlasterror()

Arguments

returns

a string containing an error message

Details

The *bluefusion.getlasterror* function is a member of the *bluefusion* class. It returns an error message describing the most recent error encountered by a *bluefusion* object. The error may have occurred during invocation of any other *bluefusion* member function.

A best practice for programmers is to check the result code for each *bluefusion* call, and if a result code indicates failure, use *bluefusion.getlasterror* to retrieve the associated error message.

Examples

The following statements illustrate the *bluefusion.getlasterror* function:

```
bluefusion bf
 integer rc
 string errmsg
 bf = bluefusion_initialize()
 rc = bf.loadqkb("XXXXX")
 // an invalid locale name -- this will cause an error

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

bluefusion.identify Function

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

Category: Blue Fusion

Syntax

bluefusion.identify(ident_def, input, result)

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

returns

Boolean [1 = success, 0 = error]

Details

The *bluefusion.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 *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized using a call to the function *bluefusion_initialize*. The member function *bluefusion.loadqkb* must then be called to load the contents of a QKB into memory and link that QKB with the *bluefusion* object. The *bluefusion* object then retains information about the QKB and the QKB locale setting.

When calling *bluefusion.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 may 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.

Examples

The following statements illustrate the *bluefusion.identify* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.loadqkb("ENUSA")
 bf.identify("Individual/Organization", "John Smith", output)
 // outputs "INDIVIDUAL"

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

bluefusion_initialize Function

Instantiates and initializes a *bluefusion* object.

Category: Blue Fusion

Syntax

bluefusion_initialize()

Arguments

returns

an initialized instance of a Blue Fusion object

Details

The *bluefusion_initialize* instantiates and initializes a *bluefusion* object. The object may then be used to invoke *bluefusion* class functions.

Examples

The following statements illustrate the *bluefusion_initialize* function:

```
bluefusion bf
 bf = bluefusion_initialize()
```

bluefusion.loadqkb Blue Fusion Function

Loads definitions from a QKB into memory and links those definitions with the *bluefusion* object.

Category: Blue Fusion

Syntax

bluefusion.loadqkb(locale)

Arguments

locale

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

returns

Boolean [1 = success, 0 = error]

Details

The function *bluefusion.loadqkb* is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized through a call to the function *bluefusion_initialize*. The function *bluefusion.loadqkb* may be called after the initialization.

The *bluefusion.loadqkb* function loads definitions from a QKB into memory and links those definitions with the *bluefusion* 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 *bluefusion* functions.

When calling *bluefusion.loadqkb* you must specify a locale code. This locale code is a five-character string representing the ISO codes for a 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 may be specified in each call to *bluefusion.loadqkb*. Only definitions associated with that locale will be loaded into memory. This means that support for only one locale at a time may be loaded for use by a *bluefusion* object. Therefore, in order to make use of QKB definitions for more than one locale, you must either use multiple instances of the *bluefusion* class or call *bluefusion.loadqkb* multiple times for the same instance, specifying a different locale with each call.

Examples

The following statements illustrate the *bluefusion.loadqkb* function:

```
bluefusion bf_en
 // we instantiate two bluefusion objects

 bluefusion bf_fr
 string output_en
 string output_fr
 bf_en = bluefusion_initialize()
 bf_fr = bluefusion_initialize()

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

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

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

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

bluefusion.matchcode Function

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

Category: Blue Fusion

Syntax

bluefusion.matchcode(match_def, sensitivity, input, result)

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 matchcode [possible values are 50-95] input

  a string representing the input value or input field name

result

  a string representing the output field name

returns

  Boolean [1 = success, 0 = error]

Details

The *bluefusion.matchcode* function generates a matchcode for an input string and outputs the matchcode to a field. The matchcode is a fuzzy representation of the input string. It may be used to do a fuzzy comparison of the input string to another string.

The function is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized through a call to the function *bluefusion_initialize*. The member function *bluefusion.loadqkb* must then be called to load the contents of a QKB into memory and link that QKB with the *bluefusion* object. The *bluefusion* object then retains information about the QKB and the QKB locale setting.

When calling *bluefusion.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 matchcode 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 sensitivity. The sensitivity indicates the level of fuzziness that is used when generating the matchcode. A higher sensitivity means that the matchcode will be less fuzzy (yielding fewer false positives and more false negatives in comparisons), while a lower sensitivity means that the matchcode will be less fuzzy (yielding fewer false negatives and more false positives in comparisons). The valid range for the sensitivity parameter is 50-95.

Examples

The following statements illustrate the *bluefusion.matchcode* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.loadqkb("ENUSA")
 bf.matchcode("Name", 85, "John Smith", output)
 // Outputs matchcode "4B~2$$$$$$$C@P$$$$$$"

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

bluefusion.pattern Function

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

Category: Blue Fusion

Syntax

bluefusion.pattern(pattern_def, input, result)

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

returns

Boolean [1 = success, 0 = error]

Details

The *bluefusion.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 may be used to perform pattern frequency analysis for a set of text strings.

The function is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized through a call to the function *bluefusion_initialize*. The member function *bluefusion.loadqkb* must then be called to load the contents of a QKB into memory and link that QKB with the *bluefusion* object. The *bluefusion* object then retains information about the QKB and the QKB locale setting.

When calling *bluefusion.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.

Examples

The following statements illustrate the *bluefusion.pattern* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.loadqkb("ENUSA")
 bf.pattern("Character", "abc123", output)
 // Outputs "aaa999"
```

bluefusion.standardize Function

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

Category: Blue Fusion

Syntax

bluefusion.standardize(stand_def, input, result)

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

returns

Boolean [1 = success, 0 = error]

Details

The *bluefusion.standardize* function generates a normalized standard for an input string and outputs the standard to a field.

The function is a member of the *bluefusion* class. A *bluefusion* object may be declared as a variable and must then be initialized through a call to the function *bluefusion_initialize*. The member function *bluefusion.loadqkb* must then be called to load the contents of a QKB into memory and link that QKB with the *bluefusion* object. The *bluefusion* object then retains information about the QKB and the QKB locale setting.

When calling *bluefusion.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.

Examples

The following statements illustrate the *bluefusion.standardize* function:

```
bluefusion bf
 string output
 bf = bluefusion_initialize()
 bf.loadqkb("ENUSA")
 bf.standardize("Name", "mcdonald, mister ronald", output)
 // Outputs "Mr Ronald McDonald"
```

Boolean Functions

Boolean is a basic data type representing a true or false value.

Boolean variables can be declared in the following formats:

```
boolean b
b=true //sets boolean b to true
b='yes' //also sets boolean b to true
b=0 //sets boolean b to false
```

This data type is used when comparisons are made. Using AND or OR in an expression also results in a boolean value.

Database Functions

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.

- dbconnect
- dbdatasources

Overview of the dbconnect Object

The *dbconnect* object allows 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 Objects

When objects are set to null they are released. Depending on whether objects are defined as static or non-static, see Declaration of Symbols for additional details. When symbols are automatically reset to null, you need to use the release() methods to explicitly release database objects.

Global Functions

dbconnect([connect_string])

Connects to a database, returns a dbconnection object

dbdatasources()

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 Connectio Administrator)

TYPE

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

HAS_CREDENTIALS

a boolean representing if the 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
dbcursor 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()
```

DBConnection Object Methods

execute([sql_string])

Executes an SQL statement and returns the number of rows affected

tableinfo([table],[schema])

Gets a list of fields for a table; the second parameter is optional

tablelist()

Gets a list of tables

prepare([sql_string])

Prepares a statement and returns a dbstatement object

select([sql_string])

Runs SQL and returns a dbcursor object

begintransaction()

Starts a transaction

endtransaction([commit])

Ends a transaction. If *commit* is true, the transaction is committed, otherwise it is rolled back

release()

Releases the connection explicitly

DBStatement Object Methods

setparaminfo([param_index],[string_type],[size])

Sets information for a parameter. *String_type* can be *string*, *integer*, *real*, *dates*, or *boolean*. If *string_type* is string, size is the string length

setparameter([param_index],[value])

Sets the value of a parameter

execute()

Executes the statement, and returns the number of rows affected

select()

Executes the statement, and returns the results as a dbcursor

release()

Explicitly releases the statement

DBCursor Object Methods

next()

Retrieves the next record

release()

Explicitly releases the cursor

columns()

Returns the number of columns

columnname([index])

Returns the name of the specified column (0-based index)

columntype([index])

Returns the type of the specified column (0-based index)

columnlength([index])

Returns the length of the specified column (0-based index)

valuestring([index])

Returns the value of the specified column as a string (0-based index)

valuereal([index])

Returns the value of the specified column as a real (0-based index)

valueinteger([index])

Returns the value of the specified column as an integer (0-based index)

Getting Table List from Database

The following code lets you get a list of tables in a particular database.

```
//Expression
string DSN
DSN="DataFlux Sample"
string connectStr
//Preparing the connection string
connectStr = "DSN=" & DSN

DBConnection dbConn
dbConn = dbConnect( connectStr )

string tablename
string datapath
string tcatalog
string tabletype

DBCursor cursTables

//Retrieve table information in a cursor
cursTables = dbConn.tablelist();

//Iterate through the cursor
while( cursTables.next() )

begin
      datapath = cursTables.valuestring(0); tcatalog =
      cursTables.valuestring(1); tablename = cursTables.valuestring(2);
      tabletype = cursTables.valuestring(3)
      pushrow()
end
cursTables.release();
```

dbconnect Function

Connect to a data source name (DSN) and returns a DBConnection object.

Category: Database

Syntax

dbconnect([connect_string])

Arguments

connect_string

is the database connection information

Details

This function is used to connect to a database. The function returns a DBConnection object.

Note: Saved Connections can also be used as parameters to this function.

Examples

The following statements illustrate the *dbconnect* function:

```
// Declare a dbconnection variable to contain the connection
// information
dbconnection my_database

// Use the dbconnect function to set a connection to the DataFlux
// Sample database
my_database = dbconnect("DSN=DataFlux Sample")
```

dbdatasources Function

Returns a list of data sources as a dbcursor.

Category: Database

Syntax

object/array = *dbdatasources*()

Examples

The following statements illustrate the *dbdatasources* function:

```
// Declare a DBCursor variable
DBCursor cursDS

//Retrieve datasource information in a cursor
cursDS = dbdatasources()
```

Data Input Functions

The Expression Engine Language (EEL) provides built-in functions that allow 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.

- fieldcount
- fieldname
- fieldtype
- fieldvalue
- readrow
- rowestimate
- seteof
- setfieldvalue

The *fieldcount*, *fieldname*, and *fieldvalue* functions allow you to dynamically access values from the parent node without knowing the names of the incoming fields.

fieldcount Function

A way of dynamically accessing values from the parent node without knowing the names of the incoming fields. Returns the number of incoming fields.

Category: Data Input

Syntax

fieldcount()

Arguments

returns

> an integer, representing the number of incoming fields from the parent node

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.

Examples

The following statements illustrate the *fieldcount* function:

```
// 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

The *fieldname* function returns the name of a specific field output from the parent node.

Category: Data Input

Syntax

fieldname(index)

Arguments

index

> is the index into the incoming fields

returns

> a string, representing the name of a specific field output from the parent node

Examples

The following statements illustrate the *fieldname* function:

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 will be set to the length in chars, if the field type is string.

Category: Data Input

Syntax

fieldtype(index [, length])

Arguments

index

is 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

length

[optional] an integer that will contain the length of the field if the field was of type string

returns

a string representing the data type of the specified field

Details

Determines the type and, optionally, the maximum length in characters (for string fields) based upon its index in the list of fields coming from the parent node. Returns a string representation of the field type (for example, integer or date).

Examples

The following statements illustrate the *fieldtype* function:

```
// 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)

Arguments

integer

is the index into the incoming fields

Examples

The following statements illustrate the *fieldvalue* function:

Example #1:

```
// Declare a string variable to contain the third field□s 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

This is a Data Job-only function. It 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

Syntax

readrow()

Arguments

returns

a Boolean representing whether there are still values in the incoming step's data

[true = there are still rows in parent node; false = no more rows to read]

Details

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 may cause undesirable results.

Examples

The following statements illustrate the *readrow* function:

Assume 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

Syntax

rowestimate(integer)

Arguments

integer

> an integer, containing an estimate for the total numbers of rows that will be output from the step

returns

> boolean [true = success; false = error]

Details

The *rowestimate* function is employed by Data Jobs to estimate the number of records that will be output from the step.

Examples

The following statements illustrate the *rowestimate* function:

```
// 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

Syntax

seteof([return_pushrow])

Arguments

return_pushrow

boolean value; to specify whether pushed rows are still returned

returns

boolean [true = success; false = error]

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.

Examples

The following statements illustrate the *seteof* function:

```
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)

Arguments

integer

is the index into the incoming fields

any

is the value you wish to set the field to

Details

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

Examples

The following statements illustrate the *setfieldvalue* function:

```
// Declare a hidden integer for the for loop, initializing it to 0, and a
hidden date field
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
```

Date and Time Functions

Dates, along with integers, reals, booleans, and strings, are considered basic data types in Expression Engine Language (EEL). Similar to other basic data types, EEL provides functions to perform operations on dates.

- formatdate
- today

formatdate Function

Returns a date formatted as a string.

Category: Date and Time

Syntax

formatdate(datevar,format)

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 returns

a string with the date formatted as a string

Formats

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 propercase
- MMM: abbreviated three-letter month
- MM: two-digit month
- DD: two-digit day
- hh: hour
- mm: minute
- ss: second

Details

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, #12-February-2010#).

Examples

The following statements illustrate the *formatdate* function:

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: February 12, 2010 12 Feb 2010

Example #3

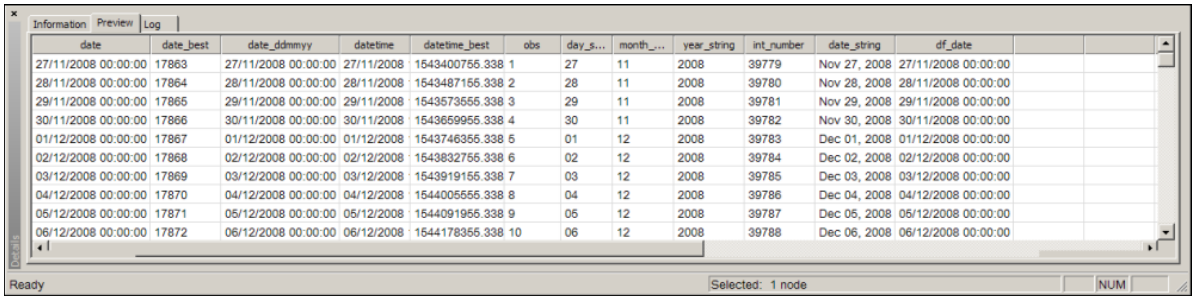
```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval, "MMMM DD, YYYY")
```

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';
```

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 date and time.

Category: Date and Time

Syntax

today()

Arguments

returns

a date that represents the current date and time value

Details

The *today* function returns the current date and time value. For example, at 4:00 pm 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.

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

Examples

The following statements illustrate the *today* function:

```
// 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()
```

Event Functions

The *raiseevent* function is available for the Expression Engine Language (EEL).

- raiseevent

raiseevent Function

Raise the specified event, pass an arbitrary number of key value pairs for event data.

Category: Event

Syntax

raiseevent(string)

Arguments

returns

Boolean value; true if the event was successfully raised; false otherwise

Details

The *raiseevent* function raises an event in the workflow engine (only). The first parameter is the name of the event to raise. There is another workflow 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. So for example, you might have the function with three parameters (name, key, value) or five parameters (name, key, value, key, value) and so on.

Examples

The following statements illustrate the *raiseevent* function:

```
raiseevent("nodeevent")
```

Execution Functions

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

- pushrow
- setoutputslot
- sleep

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

Syntax

pushrow()

Arguments

returns

boolean (true = success; false = error)

Details

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. When the stack is empty, the rows are read from above as usual. This function always returns true.

Examples

The following statements illustrate the *pushrow* function:

```
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

Set output slot to slot. This will become the output slot when the expression exits.

Category: Execution

Syntax

setoutputslot(slotnumber)

Arguments

slotnumber

an integer representing the exit slot

returns

boolean [true = success; false = error]

Details

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

Examples

The following statements illustrate the *setoutputslot* function:

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

sleep Function

Use the *sleep* function to set a sleep time at the specified number of milliseconds and invokes the interrupt handler.

Category: Execution

Syntax

sleep(integer duration)

Arguments

duration

an integer representing the number of seconds to pause

returns

boolean value [true = successful; false = unsuccessful]

Details

The *sleep* function causes the job to pause for the specified number of seconds

Examples

The following statements illustrate the *sleep* function:

```
sleep(5)
```

Results: The job sleeps for 5 seconds

External File Functions

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.

- close
- copyfile
- deletefile
- execute
- filedate
- fileexists
- movefile
- open

- position
- readbytes
- readline
- seekbegin
- seekcurrent
- seekend
- writebytes
- writeline

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 may 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®, type:

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

Or, to execute from the UNIX® shell, type:

```
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"

close Function

Closes an open file.

Category: External file

Syntax

fileobject.*close*(<filename>)

Arguments

filename

> [optional] a string representing the name of the file to be closed, this can be specified as a fixed string, field name, or expression

returns

> Boolean [1 = success, 0 = error]

Details

The *close* method closes the file that is provided in the filename parameter. If the filename parameter is blank, the currently open file (which was opened by using a fileobject.*open* call) will be closed.

Examples

The following statements illustrate the *close* function:

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

copyfile Function

Copies a file.

Category: External file

Syntax

copyfile(source_file, target_file)

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

returns

> Boolean [true = success, false = error]

Details

The *copyfile* function copies a file. If the target file exists, it will overwrite it.

Examples

The following statements illustrate the *copyfile* function:

```
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 a file.

Category: External file

Syntax

deletefile(filename)

Arguments

filename

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

returns

> Boolean [true = success, false = error]

Details

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

Examples

The following statements illustrate the *deletefile* function:

```
string filename
boolean rc_ok

filename="C:\mydata_copy.txt"

rc_ok = deletefile(filename)
```

execute Function

Executes a program.

Category: External file

Syntax

execute(filename<option1, option2,..., <option N>)

Arguments

filename

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

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

returns

> an integer, which is the existing status of the program that was executed. If an error occurs, such as the program not found then -1 is returned

Details

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

Examples

The following statements illustrate the *execute* function:

```
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

Syntax

filedate(filename<datetype>)

Arguments

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

datetype

[optional] 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]

returns

a 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 parameter datetype is omitted the function behaves like the value would have been specified as false.

Examples

The following statements illustrate the *filedate* function:

```
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 file exists.

Category: External file

Syntax

fileexists(filename)

Arguments

filename

> a string representing the name of the file for which the existence is to be checked. This can be specified as a fixed string, field name, or expression

returns

> Boolean [true = success, false = error]

Details

The *fileexists* function returns true if the file exists.

Examples

The following statements illustrate the *fileexists* function:

```
string filename
boolean rc_ok

filename="C:\doesexist.txt"

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

movefile Function

Moves or renames a file.

Category: External file

Syntax

movefile(old_file_name, new_file_name)

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 will be moved. This can be specified as a fixed string, field name, or expression

returns

> Boolean (true = success, false = error)

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 will not be moved and false will be returned.

Examples

The following statements illustrate the *movefile* function:

```
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 file.

Category: External file

Syntax

fileobject.*open*(filename<openmode>)

Arguments

filename

> a string representing the name of the file to be opened. If the file does not exist it will be 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]

returns

> a boolean [1 = success, 0 = error]

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 will be created. If the openmode is not specified a value of false will be returned.

When openmode is "a", the *writebytes* and *writeline* methods will write at the end of the file, unless *seekbegin*, *seekcurrent*, or *seekend* methods are used to adjust the position in the file. In that case the information will be 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.

Examples

The following statements illustrate the *open* function:

```
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

Syntax

fileobject.*position*()

Arguments

returns

> an integer representing the current position (offset) from the beginning of the file

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.

Examples

The following statements illustrate the *position* function:

```
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

Syntax

fileobject.*readbytes*(number_of_bytes, buffer)

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 will contain the bytes that were read. This parameter can be specified as a fixed string, field name, or expression

returns

Boolean [1 = success, 0 = error]

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 will be put into the buffer. It returns true on success otherwise false is returned.

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 will also read EOL characters. So when reading a windows text file like this:

C:\filename.txt
abc
def

A *readbytes*(7, buffer) statement will cause the field buffer to contain the following value "abc\n de", which consists of all the info from the first line (3 bytes), followed by a CR and a LF character (2 bytes) which is represented by "\n", followed by the first 2 bytes from the second line. To read text files use the *readline*() method.

Examples

The following statements illustrate the *readbytes* function:

```
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

Syntax

fileobject.*readline*()

Arguments

returns

> a string containing the line that was read from the file

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.

Examples

The following statements illustrate the *readline* function:

```
file f
string input

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

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

Syntax

fileobject.*seekbegin*(position)

Arguments

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

returns

> Boolean (1 = success, 0 = error)

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 1 means that reading starts after the 1st position in the file.

Examples

The following statements illustrate the *seekbegin* function:

```
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

Syntax

fileobject.*seekcurrent*(position)

Arguments

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

returns

> Boolean (1 = success, 0 = error)

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.

Examples

The following statements illustrate the *seekcurrent* function:

```
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

Syntax

fileobject.*seekend*(position)

Arguments

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

returns

> Boolean (1 = success, 0 = error)

Details

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

You can use this method when writing information at the end of the file.

You can also use this method in combination with the *position*() method to determine the size of a file.

Examples

The following statements illustrate the *seekend* function:

```
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

Syntax

fileobject.*writebytes*(number_of_bytes, buffer)

Arguments

number_of_bytes

> an integer specifying the number of bytes that will be 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

returns

> an integer representing the number of bytes written

Details

The *writebytes* method writes the specified number of bytes to a file starting at the current position in the file. This method will overwrite 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 will be increased.

When buffer is larger than number_of_bytes specified then only the first number_of_bytes from buffer will be 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.

Examples

The following statements illustrate the *writebytes* function:

```
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

Write a line to a file.

Category: External file

Syntax

fileobject.*writeline*(string)

Arguments

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

returns

> a boolean [1 = success, 0 = error]

Details

The *writeline*() method writes the string at the current position in the file. This method will overwrite 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 will be increased.

The file needs to be opened in write or append mode for this method to work.

Examples

The following statements illustrate the *writeline* function:

```
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()
```

Information/Conversion Functions

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

- isalpha
- isblank
- isnull
- isnumber
- locale
- toboolean
- typeof

isalpha Function

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

Category: Information/Conversion Functions

Syntax

isalpha (in_string)

Arguments

string

a string of characters to be searched for any alphabetic characters

returns

Boolean value; true if the "in_string" contains only alpha characters; false otherwise

Details

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

Examples

The following statements illustrate the *isalpha* function:

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

The *isblank* function checks to see if an argument contains a blank, empty value. When the argument value is blank, the function returns true, otherwise it returns false.

Category: Information/Conversion Functions

Syntax

boolean *isblank* (argvalue)

Arguments

argvalue

string, date, integer, real

returns

Boolean [true = successful, false = failed]

Details

The *isblank* function takes the following argument types: string, date integer, real.

Examples

The following statements illustrate the *isblank* function:

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

y="Hello"

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

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

if( isblank(z) )
   error_message3 =" Date z value is blank"
else
   error_message3 =" Date z value is not blank"
```

isnull Function

The *isnull* function checks to see if an argument value contains a null value. When the argument value is null, the function returns true, otherwise it returns false.

Category: Information/Conversion Functions

Syntax

boolean *isnull*(argvalue)

Arguments

argvalue

string, date, integer, real

returns

Boolean [true = successful, false = failed]

Details

The *isnull* function takes the following argument types: string, date integer, real.

Examples

The following statements illustrate the *isnull* function:

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

The *isnumber* function checks to see 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/Conversion Functions

Syntax

boolean *isnumber*(argvalue)

Arguments

argvalue

string, date, integer, real

returns

Boolean [true = successful, false = failed]

Details

The *isnumber* function takes the following argument types: string, date integer, real.

Examples

The following statements illustrate the *isnumber* function:

```
string x
string y
date 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"
```

locale Function

Sets the locale, which affects certain operations such as uppercasing and date operations. It will return the previous locale. If no parameter is passed, the current locale is returned. The locale setting is a global setting.

Category: Information

Syntax

string *locale*()

string *locale*("locale_string")

Arguments

returns

string value of the current locale setting

Details

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

Examples

The following statements illustrate the *locale* function:

```
string currentSetting
string newSetting

currentSetting = locale();

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

toboolean Function

Converts the argument to a boolean value.

Category: Information/Conversion

Syntax

boolean *toboolean*(value)

Arguments

value

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

returns

a Boolean value is returned if value can be converted to a Boolean value

Examples

The following statements illustrate the *toboolean* function:

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

typeof Function

Identifies the data type of the passed in value.

Category: Information/Conversion Functions

Syntax

string *typeof*(in_value)

Arguments

returns

string identifier for the value type:

boolean values return boolean

integer values return integer

real values return real

string values return string

Details

The *typeof* Function identifies the data type of in_value.

Examples

The following statements illustrate the *typeof* function:

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
```

Logging Functions

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

- logmessage
- print
- raiseerror
- sendnodestatus

logmessage Function

The *logmessage* function prints a message to the log.

Category: Logging

Syntax

boolean *logMessage*(string message)

Details

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

Examples

The following statements illustrate the *logmessage* function:

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

print Function

Prints the string to the step log. If the second parameter is true, no linefeeds will be appended after the text.

Category: Logging

Syntax

print(string[, no linefeed])

Arguments

string

> is the text to be printed, this can be specified as a text constant or a field name

no linefeed

> [optional] the second boolean determines whether or not linefeeds will be printed to the log after the text

returns

> boolean [true = success; false = error]

Examples

The following statements illustrate the *print* function:

```
// 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

The *raiseerror* function prints a message to the run log.

Category: Logging

Syntax

boolean *raiseerror*(string usererror)

Arguments

usererror

> is a string that is printed to the jobs output

returns

> boolean value

Details

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 see if the associated condition was responsible for stopping the job.

Examples

The following statements illustrate the *raiseerror* function:

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

sendnodestatus Function

The *sendnodestatus* function sends a node status.

Category: Logging

Syntax

boolean *sendnodestatus*(key, status)

Arguments

returns

> boolean value; true if the *sendnodestatus* was successful; false otherwise

Details

The *sendnodestatus* function tells the workflow 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)
```

Examples

The following statements illustrate the *sendnodestatus* function:

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

Macro Variable Functions

Macros (or variables) are used to substitute values in a job. This may be useful if you want to run a job in different modes at different times. For example, you may want to have the job run 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 using another method.

- getvar
- setvar
- varset

Using Macro 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 BLUEFUSION/QKB. To use a macro in a job, enter it with double percent signs before and after the value, for example:

Old value:

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

Using macros, you enter instead:

```
%%MYFILE%%
```

You can also use the macro to substitute some part of the parameter, for 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, you can go to the Advanced tab and enter it there. After you have entered a macro under the Advanced tab, you get a warning if you try to return to the standard property dialog. Depending on your macro, you may need to avoid the standard property dialog and use the advanced dialog thereafter. If the property value is plain text, you can return to the standard dialog.

You can choose to use variables for data input paths and filenames in the Data Job Expression node. You can declare macro variables by entering them in the Macros folder of the Administration riser bar, by editing the macro.cfg file directly, or by 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 choose to 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.

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:

```
-VAR "key1=value1,key2=value2"
-VARFILE "C:\mymacros.txt"
```

Note: The return code can be checked by creating a batch file, then checking the errorlevel in the batch file by using:

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

If the return code is not set, it will return a zero on success and one 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 DataFlux 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, which 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.

Predefined Macro	Description

_RETURN_CODE	Sets the return code for dfwfproc.
	
	`// Set the return code to 100`
	`setvar("_RETURN_CODE",100)`

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 may not be reflected in nodes on that page. This is because those nodes have already read the old value of the macro and may have acted upon it (such as opening a file before the macro value was changed). This issue arises only from using *setvar*(), and thus *setvar*() is useful only for setting values that are read on following pages, or from other expression nodes with *getvar*().

getvar Function

Returns runtime variables. These are variables that are passed into DataFlux Data Management Studio on the command line using -VAR or -VARFILE.

Category: Macro Variable

Syntax

getvar(string[, string])

Arguments

string

the first parameter is the name of the variable and is case insensitive. The second parameter is the value that is returned if the variable does not exist

Details

Variables that are passed into DataFlux Data Management Studio on the command line using -VAR or -VARFILE.

Examples

The following statements illustrate the *getvar* function:

```
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. Returns true.

Category: Macro Variable

Syntax

setvar(string, sting)

Arguments

string

value indicated by the first parameter, to the values in the second parameter.

Examples

The following statements illustrate the *setvar* function:

```
string xyz
xyz=getvar("myvar3")
```

varset Function

Sets the Data Job macro variable value specified by the first parameter. Returns true.

Category: Macro Variable

Syntax

varset(macroname, value)

Arguments

macroname

Sets the macro named in macroname to value. The macroname argument is a string

value

The value argument can be any of the following: real, integer, string, date, or boolean

returns

Boolean value; true if macro was set to new value

Examples

The following statements illustrate the *varset* function:

```
varset("myMacroName", "newValue");
```

Mathematical Functions

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

- abs
- ceil
- floor
- max
- min
- pow
- round

abs Function

Returns the absolute value of a number.

Category: Mathematical

Syntax

abs(argument)

Arguments

argument

a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents the absolute value of the argument

Details

The *abs* function returns a nonnegative number that is equal in magnitude to the magnitude of the argument.

Examples

The following statements illustrate the *abs* function:

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

ceil Function

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

Category: Mathematical

Syntax

ceil(argument)

Arguments

argument

a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents the smallest integer that is greater than or equal to the argument

Details

The *ceil* function returns the smallest integer that is greater than or equal to the argument, also called rounding up (ceiling).

Examples

The following statements illustrate the *ceil* function:

```
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

Syntax

floor(argument)

Arguments

argument

a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents the largest integer that is less than or equal to the argument

Details

The *floor* function returns the largest integer that is less than or equal to the argument, also called rounding down.

Examples

The following statements illustrate the *floor* function:

```
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

Syntax

max(argument1 <, argument2,…,argument N>)

Arguments

argument1

a real that can be specified as a numeric constant, field name, or expression

argument2..N

[optional] a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents the maximum value of the series of input values

Details

The max function returns the maximum value of a series of values. The function returns null if all values are null.

Examples

The following statements illustrate the *max* function:

```
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

Syntax

min(argument1 <, argument2, …,argument N>)

Arguments

argument1

a real that can be specified as a numeric constant, field name, or expression

argument2..N

[optional] a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents the minimum value of the series of input values

Details

The *min* function returns the maximum value of a series of values. The function returns null if all values are null.

Examples

The following statements illustrate the *min* function:

```
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

Syntax

pow(x, y)

Arguments

x

a real that can be specified as a numeric constant, field name, or expression

y

a real that can be specified as a numeric constant, field name, or expression

returns

a real that represents x raised to the power y (x^y)

Details

The *pow* function raises x to the power y (x^y).

Examples

The following statements illustrate the *pow* function:

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

round Function

The *round* function rounds a number to the nearest number with the specified decimal places.

Category: Mathematical

Syntax

round(argument <decimals>)

Arguments

argument

a real that can be specified as a numeric constant, field name, or expression

decimals

[optional] specifies a numeric constant, field name, or expression representing the amount of decimals to provide in the result of the rounding operation

returns

a real that represents the argument rounded to the nearest number with the specified decimal places

Details

The *round* function rounds the argument to the nearest number with the specified number of decimals. If no value is provided for decimals, 0 is assumed. 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.

Examples

The following statements illustrate the *round* function:

```
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
```

Regular Expression Functions

The regular expression (regex) object allows you to perform regular expression searches of strings.

- compile
- findfirst
- findnext
- matchlength
- matchstart
- replace
- substringcount
- substringlength
- substringstart

compile Function

This function compiles a valid regular expression using the encoding indicated.

Category: Regular Expression

Syntax

r.*compile*(regex, encoding)

Arguments

regex

is a Perl-compatible regular expression

encoding

> is a string that defines the encoding constant shown in the table below. Use a value from the Encoding column (see Encoding). This input is optional. Not including this parameter instructs DataFlux to use the default encoding for the operating system

returns

> 1 (true) if the regular expression compilation was successful, 0 (false) if regular expression compilation failed. Failure could be due to an incorrectly formatted regular expression or possibly an invalid encoding constant

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 use 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), determine how long the matching patterns are, and to replace patterns where desired.

For performance reasons it is best to compile a regular expression in a pre-processing step in the expression node. This means that the regular expression will be 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 case you many need to have the regular expression compiled before every row is evaluated. For example, you can use a variable to define the regular expression you want to compile. The variable may come from the data row itself and you would need to recompile the regular expression for each row have the pattern searching work correctly.

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

Examples

The following statements illustrate the *compile* regex function:

> **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()
```

findfirst Function

The *findfirst* function searches the specified string for a pattern match using an already compiled regular expression.

Category: Regular Expression

Syntax

r.*findfirst*(input)

Arguments

input

> is a string value in which you would like to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue") or this can 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')
>
> **Restriction:** Must be non-null and not blank

returns

> Boolean: 1 (true) if a pattern match was found, 0 (false) if no pattern match was found

Details

The *findfirst* function will indicate if one or more pattern matches were found in the input. This function can be used to enter a logical loop that will pull out a series of matched patterns from an input string. A false value indicates that no match is found and that processing can continue.

Examples

The following statements illustrate the *findfirst* regex function:

> **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 typed 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

This function continues searching the string for the next match after using the *findfirst*() function.

Category: Regular Expression

Syntax

r.*findnext*(input)

Arguments

input

> is a string value in which you would like to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue") or this can 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')
>
> **Restriction:** Must be non-null and not blank

returns

> Boolean: 1 (true) if a pattern match was found, 0 (false) if no pattern match was found

Details

The *findnext* function will indicate that another pattern match has been found after *findfirst*() has been used. Using a "While" statement loop will let you iterate through all potential pattern matches using this function as long the return value is equal to true.

Examples

The following statements illustrate the *findnext* regex function:

> **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 typed 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

The *matchlength* function returns the length of the last pattern match found.

Category: Regular Expression

Syntax

r.*matchlength*()

Arguments

There is no input value. This function operates on the pattern match substring found using *findfirst*() or *findnext*().

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.

Details

The *matchlength* function should be used 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.

Examples

The following statements illustrate the *matchlength* regex function:

> **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 typed 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

The *matchstart* function returns the location of the last pattern match found.

Category: Regular Expression

Syntax

r.*matchstart*()

Arguments

input

> is a string value in which you would like to search for the pattern defined by your compiled regular expression
>
> **Restriction:** Must be non-null and not blank

returns

> an integer value that represents the starting character position of the substring 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

Details

The *matchstart* function returns the position of a substring that has been matched to the regular expression. A logical loop can be used to iterate through all matching substrings and 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.

Examples

The following statements illustrate the *matchstart* regex function:

> **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 typed 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 agin
         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

Syntax

r.*replace*(input, replacement value)

Arguments

input

> is a string value in which you would like to search for the pattern defined by your compiled regular expression. This can be an explicit string ("MyValue") or this can 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')
>
> **Restriction:** Must be non-null and not blank

replacement value

> is a string value that will take the place of the substring matched by the compiled regular expression

returns

> 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

Details

This *replace* regular expression 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 a two hyphens with any letter in between (-A-, -B-, etc) and replace with a single letter (Z for instance), you would compile your regular expression for finding the hyphen/letter/hyphen pattern, and then use "Z" as the replacement value of the *replace*() function passing in a variable or string value for input.

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 allows you to use captured substrings in replacement values.

Examples

The following statements illustrate the *replace* regex function:

> **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 typed 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

The *substringcount* function returns the number of subpatterns found to have matched the pattern specified by the compiled regular expression. The regular expression has to contain subpatterns that can be pattern matched.

Category: Regular Expression

Syntax

r.*substringcount*()

Arguments

There is no input value.

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

Details

The *substringcount* function should be used to find the total number of subpatterns found to have matched the regular expression. Normally simple regular expressions will evaluate to "1" but if you design regular expressions using subpatterns then this function will return the number found.

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

> (Mr|Mrs) Smith

Here the subpattern is the "(Mr|Mrs)" and using this function will return the number "2" for the count of substrings since the entire string is considered the first subpattern and the part inside the parentheses is the second subpattern.

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

Examples

The following statements illustrate the *substringcount* regex function:

> **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 typed 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 Function

The *substringlength* function returns the length of the nth captured subpattern. The regular expression must contain subpatterns that can be pattern matched.

Category: Regular Expression

Syntax

r.*substringlength*(nth)

Arguments

nth

> is a positive integer value that specifies the substring whose length you want to be returned
>
> **Restriction:** Must be non-null and not blank

returns

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

Details

The *substringlength* can be used to find the length in characters of the subpattern specified by passing an integer value as input. See the Details for the function *substringcount*() for more information on working with subpatterns.

Most simple regular expressions will not have subpatterns and this function will behave similarly to *matchlength*(). However if your regular expression does use subpatterns then this function can be used to find the length of individually captured subpatterns found within the overall matched pattern.

Examples

The following statements illustrate the *substringlength* regex function:

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 typed 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

The *substringstart* function returns the start location of the nth captured subpattern. The regular expression has to contain subpatterns that can be pattern matched.

Category: Regular Expression

Syntax

r.*substringstart*(nth)

Arguments

nth

> is a positive integer value that specifies the subpattern whose starting location you would like to be returned
>
> **Restriction:** Must be non-null and not blank

returns

> a positive integer value that represents the starting character location of a matched subpattern. NULL is returned if there is no substring currently under consideration and therefore no length to return

Details

The *substringstart* takes the input integer you supply and returns a starting location for the subpattern represented by that input integer. Use *substringcount*() to determine the number of subpatterns under consideration. Use *substringlength*() with this function to pull out the matched subpatterns and use them in evaluation logic of your expression code.

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

Examples

The following statements illustrate the *substringstart* regex function:

> **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
```

Search Functions

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

- inlist

inlist Function

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

Category: Search

Syntax

inlist (target_parameter1, value_parameter1, value_parameter2)

Arguments

target_parameter

string, integer or date value; this value will be "searched" value_parameter

value_parameter

string, integer or date value; this value will be searched for amongst the check_parameter arguments

returns

Boolean [true = successful, false = failed]

Details

The first parameter is compared against each of the following value_parameters. True is returned if a match is found.

Examples

The following statements illustrate the *inlist* function:

```
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)
```

String Functions

There are several functions available in Expression Engine Language (EEL) that affect the built-in string data type.

- aparse
- asc
- chr
- compare

- edit_distance
- has_control_chars
- instr
- left
- len
- lower
- match_string
- mid
- mkdir
- parse
- pattern
- replace
- right
- rmdir
- sort
- sort_words
- todate
- tointeger
- toreal
- tostring
- trim
- upper
- username
- vareval

aparse Function

Parses a string into words and returns the number of words found.

Category: String

Syntax

aparse(string,delimiter,word_list)

Arguments

string

> a string that represents 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 runtime error

delimiter

> 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

> a string array that represents the words that were found during parsing, this is specified as a field name

returns

> an integer that represents the number of words found

Details

The *aparse* function fills the parameter word_list with a list of words using the delimiter specified. The function returns the number of words found.

If parameter string is empty ("") a value of 1 is returned and the word_list has one element that contains an empty string.

The *parse* function is very similar. It returns individual string fields instead of a string array, those 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.

Examples

The following statements illustrate the *aparse* function:

```
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

Syntax

asc(string)

Arguments

string

> a string that represents 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

returns

> an integer that represents the position of a character in the ASCII collating sequence

Details

The *asc* function returns the position of the specified character in the ASCII collating sequence.

See Appendix A: ASCII Values for a complete list of ASCII values.

Examples

The following statements illustrate the *asc* function:

```
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

Syntax

chr(n)

Arguments

n

> an integer that represents a specific ASCII character, this can be specified as a numeric constant, a field name, or an expression

returns

> a string that represents the n-th character in the ASCII collating sequence

Details

The *chr* function returns n-th character in the ASCII collating sequence.

See Appendix A: ASCII Values for a complete list of ASCII values.

Examples

The following statements illustrate the *chr* function:

```
character_content = chr(97) // outputs the letter "a"
ascii_value = asc("a") // outputs 97
```

compare Function

Returns the result of comparing two strings.

Category: String

Syntax

compare(string1, string2<,modifier>)

Arguments

string1

a string to be used in the comparison, this can be specified as string constant, field name, or expression

string2

a string to be used in the comparison, this can be specified as string constant, field name, or expression

modifier

[optional] a boolean string that modifies the action of the *compare* function, this can be specified as string constant, field name, or expression [true = case insensitive, false = case sensitive]

returns

an integer representing the result of a lexicographical comparison of the two strings:

```
[-1 = string1 < string 2,
0 = string1 equals string2,
1 = string1 > string2]
```

Details

The *compare* function compares two strings lexicographically. If the parameter modifier is omitted the function behaves like a value of false was specified.

If you just want to check whether two strings are equal it is more efficient to use the == operator, for example:

```
if string1 == string2 then match=true
```

The *match_string* function can be used to do string comparisons using wildcards.

Examples

The following statements illustrate the *compare* function:

```
// 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

Syntax

edit_distance(string1,string2)

Arguments

string1

> a string to be used in the comparison, this can be specified as string constant, field name, or expression

string2

> a string to be used in the comparison, this can be specified as string constant, field name, or expression

returns

> an integer representing the number of corrections that would need to be applied to turn string1 into string2

Details

The *edit_distance* function returns the number of corrections that would need to be applied to transform string1 into string2.

Examples

The following statements illustrate the *edit_distance* function:

```
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 if the string contains control characters.

Category: String

Syntax

boolean *has_control_chars*(string targetstring)

Arguments

targetstring

is a string to be search for existence of any ASCII control characters

returns

boolean [true = successful; false = failed]

Details

The *has_control_chars* function can be used to identify non-printable ASCII controls characters as found on the ASCII character table.

Examples

The following statements illustrate the *has_control_chars* function:

```
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

Syntax

instr(source,excerpt<,count>)

Arguments

source

> a string that represents the string of characters to search for in source, this can be specified as string constant, field name, or expression

count

> [optional] an integer that specifies 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

returns

> an integer representing the position at which the excerpt was found

Details

The *instr* function searches source from left to right for the count-th occurrence of excerpt. If the string is not found in source, the function returns a value of 0.

Examples

The following statements illustrate the *instr* function:

```
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 leftmost characters of a string.

Category: String

Syntax

left(source,count)

Arguments

source

> a string to be searched, this can be specified as string constant, field name, or expression

count

> an integer that specifies how many characters to return, this can be specified as numeric constant, field name, or expression

returns

> a string representing the leftmost characters

Details

The *left* function returns the leftmost count characters of source.

When a count of zero or less is specified an empty string is returned. If source is null the function returns a null value.

Examples

The following statements illustrate the *left* function:

```
source = "abcdefg"
result = left(source, 4) // outputs the string "abcd"
```

len Function

Returns the length of a string.

Category: String

Syntax

len(source)

Arguments

source

> a string for which the length needs to be determined, this can be specified as string constant, field name, or expression

returns

> an integer representing the length of the string

Details

The *len* function returns the length of a string

The length of an empty string ("") is zero. If source is null the function returns a null value. To remove leading and trailing blanks use the *trim* function.

Examples

The following statements illustrate the *len* function:

```
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

Syntax

lower(source)

Arguments

source

a string, this can be specified as string constant, field name, or expression

returns

a string, representing the source string in lowercase

Details

The *lower* function returns the lowercase of a string. If source is null the function returns a null value.

Examples

The following statements illustrate the *lower* function:

```
source = "MÜNCHEN in Germany"
lowcase_string = lower(source) // outputs "münchen in Germany"
```

match_string Function

Determines if the first string matches the second string, which can contain wildcards.

Category: String

Syntax

match_string(string1,string2)

Arguments

string1

> a string that needs to be searched, this can be specified as string constant, field name, or expression

string2

> a string that represents a search pattern, this can be specified as string constant, field name, or expression

returns

> a boolean, indicating whether a match has been found [true = match was found; false = no match was found]

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).

Question marks may be used as a wildcard but will only be matched 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, regular expressions instead of the *match_string*() function are recommended.

If source is null the function returns a null value.

Examples

The following statements illustrate the *match_string* function:

```
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

Syntax

mid(source,position<,length>)

Arguments

source

> a string that needs to be searched, this can be specified as string constant, field name, or expression

position

> an integer that represents the beginning character position, this can be specified as a numeric constant, field name, or expression

length

> [optional] an integer that represents the length of the substring to extract, this can be specified as a numeric constant, field name, or expression

returns

> a string, representing the extracted substring

Details

The *mid* function returns the substring with specified length starting at specified position.

If length is null, zero, or larger than the length of the expression that remains in source after position, the remainder of the expression will be returned.

If length is omitted, the remainder of the string will be extracted.

Examples

The following statements illustrate the *mid* function:

```
source = "06MAY98"

result = mid(source, 3, 3) // outputs "MAY"
result = mid(source, 3) // outputs "MAY98"
```

mkdir Function

Creates a directory. If the second parameter is true, the directory will be recursively created.

Category: String

Syntax

boolean *mkdir*(string[, boolean])

Arguments

boolean

> is a boolean expression that is either true or false

string

> is the text string that contains the directory to be created

boolean

> [optional] the second boolean expression will create the directory recursively based on whether or not the expression is true or false

Examples

The following statements illustrate the *mkdir* function:

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 for Boolean b to "true"
b=true

// Use the MKDIR function to create the new directory, recursively based on the
value of Boolean b
d mkdir(dir,b)
```

parse Function

Parses a string into words and returns the number of words found.

Category: String

Syntax

parse(string, delimiter, word1 <, word2 , …, wordN>)

Arguments

string

> a string that represents the string that needs to be separated into words, this can be specified as fixed string, field name, or expression

delimiter

> 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

word1

> a string that represents the first word found, this is specified as a field name

word2...wordN

> [optional] a string that represents the words found, this is specified as field names

returns

> an integer that represents the number of words found

Details

The *parse* function fills the provided parameters with words using the delimiter specified. The function returns the number of words found.

If string is null the function returns null. If string is empty ("") a value of 1 is returned.

The *aparse* function is similar. It is more flexible, as you do not have to know, in advance, what the maximum number of words is and can be used to easily determine the last word in a string.

Examples

The following statements illustrate the *parse* function:

```
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 if a string has numbers or uppercase and lowercase characters.

Category: String

Syntax

pattern(string)

Arguments

string

> a string that represents the string for which the pattern needs to be determined, this can be specified as fixed string, field name, or expression

returns

> a string with the pattern found

Details

The *pattern* function replaces each number with a 9, each uppercase character with an "A" and each lowercase character with an "a". Other characters are not replaced.

If string is null the function returns null. If string is empty ("") an empty value is returned.

Examples

The following statements illustrate the *pattern* function:

```
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 string with the replacement made.

Category: String

Syntax

replace(source,search,replace,string<, integer>)

Arguments

source

> a string that needs to be searched, this can be specified as string constant, field name, or expression

search

> a string that represents the text to be searched for, this can be specified as string constant, field name, or expression

replace

> a string that represents the replacement for text that was found, this can be specified as string constant, field name, or expression

count

> an integer that represents how many replacements should be made, this can be specified as numeric constant, field name, or expression

returns

> a string that represents the string with the replacements made

Details

The *replace* function replaces the first occurrence of one string with another string, and returns the string with the replacement made. If count is omitted or set to zero, all occurrences will be replaced in the string.

If source is null the function returns a null value.

Examples

The following statements illustrate the *replace* function:

```
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 rightmost characters of a string.

Category:

Syntax

right(source,count)

Arguments

source

> a string to be searched, this can be specified as string constant, field name, or expression

count

> an integer that specifies how many characters to return, this can be specified as numeric constant, field name, or expression

returns

> a string representing the rightmost characters

Details

The *right* function returns the rightmost count characters of source.

When a count of zero or less is specified an empty string is returned. If source is null the function returns a null value.

Examples

The following statements illustrate the *right* function:

```
source = "abcdefg"
result = right(source, 4) // outputs the string "defg"

source = "abcdefg "
result = right(source, 4) // outputs the string "fg "
```

rmdir Function

Deletes a directory if it is empty.

Category: String

Syntax

boolean *rmdir*(string)

Arguments

boolean

> is a boolean expression that is either true or false

string

> is the text string that contains the directory to be checked and removed if empty

Examples

The following statements illustrate the *rmdir* function:

```
// 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)
```

sort Function

Returns the string with its characters sorted alphabetically.

Category: String

Syntax

sort(source<,ascending<,remove_duplicates>>)

Arguments

source

> a string that needs to be sorted, this can be specified as string constant, field name, or expression

ascending

> [optional] a boolean that represents whether the text should be sorted in ascending order, this can be specified as boolean constant, field name, or expression

remove_duplicates

> [optional] a boolean that represents whether duplicate characters should be removed, this can be specified as boolean constant, field name, or expression

returns

> a string that represents the sorted string

Details

The *sort* function returns a string with its characters sorted alphabetically. If ascending is true or omitted, the string will be sorted in ascending order. A false value results in a descending sort order. If remove_duplicates is true duplicate characters are being discarded. If remove_duplicates is omitted duplicates are not discarded.

If source is null the function returns a null value.

Examples

The following statements illustrate the *sort* function:

```
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, consisting of the words within the input string sorted alphabetically.

Category: String

Syntax

sort_words(source<,ascending<,remove_duplicates>>)

Arguments

source

a string that needs to be sorted, this can be specified as string constant, field name, or expression

ascending

[optional] a boolean that represents whether the words in the input string should be sorted in ascending order, this can be specified as boolean constant, field name, or expression

remove_duplicates

[optional] a boolean that represents whether duplicate words should be removed, this can be specified as boolean constant, field name, or expression

returns

a string that represents the sorted string

Details

The *sort_words* function returns a string with its words sorted alphabetically. If ascending is true or omitted, the string will be sorted in ascending order. A false value results in a descending sort order. If remove_duplicates is true duplicate words are being discarded. If remove_duplicates is omitted duplicates are not discarded.

> **Note:** Special characters like ",.!" are not treated as separation characters.

If source is null the function returns a null value.

Examples

The following statements illustrate the *sort_words* function:

```
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.

Category: String

Syntax

date *todate*(any)

Arguments

date

contains the date value returned by the function

any

is the value passed into the function for conversion to a date value

Examples

The following statements illustrate the *todate* function:

```
// 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

Syntax

integer *tointeger*(any)

Arguments

integer

contains the integer value returned by the function

any

is the value passed into the function for conversion to an integer value

Examples

The following statements illustrate the *tointeger* function:

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
```

toreal Function

Converts the argument to a real value.

Category: String

Syntax

real *toreal*(any)

Arguments

real

contains the real value returned by the function

any

is the value passed into the function for conversion to a real value

Examples

The following statements illustrate the *toreal* function:

```
// 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
```

tostring Function

Converts the argument to a string value.

Category: String

Syntax

string *tostring*(any)

Arguments

string

contains the string value returned by the function

any

is the value passed into the function for conversion to a string value

Examples

The following statements illustrate the *tostring* function:

```
// 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
```

trim Function

Removes leading and trailing white space.

Category: String

Syntax

trim(source)

Arguments

source

> a string from which the leading and trailing white space needs to be removed, this can be specified as string constant, field name, or expression

returns

> a string representing the string with leading and trailing white space removed

Details

The *trim* function returns the string with leading and trailing white space removed.

If source is null the function returns a null value. If source is an empty value ("") the function returns an empty value.

Examples

The following statements illustrate the *trim* function:

```
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

Syntax

upper(source)

Arguments

source

a string, this can be specified as a string constant, field name, or expression

returns

a string representing the source string in uppercase

Details

The *upper* function returns the uppercase of a string. If source is null the function returns a null value.

Examples

The following statements illustrate the *upper* function:

```
source = "MÜNCHEN in Germany"
upcase_string = upper(source) // outputs "MÜNCHEN IN GERMANY"
```

username Function

Returns the operating system username of the logged in user.

Category: String

Syntax

string *username*()

Arguments

string

contains the string returned by the function

Examples

The following statements illustrate the *username* function:

```
// Declare the string value for the function
string user

// Use the USERNAME function to get the OS user name
user = username()
```

vareval Function

Evaluates and returns the value of a variable with the given name.

Category: String

Syntax

string *vareval*(string)

Arguments

string

the first string contains the string returned by the function

string

the second string is the field name passing into the function for conversion to a string

Details

The *vareval* function evaluates a string as though 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.

Examples

The following statements illustrate the *vareval* function:

```
// 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)
     n=n+1
     pushrow()
```

```
end
// Return false to prevent the last row from showing up twice
return false
```

Experimental Functions

The following functions are currently experimental and are not yet fully supported.

We encourage you to experiment with these functions and e-mail any questions, comments, or issues to experimental@dataflux.com.

Function	Syntax	Description
decode	integer *decode*(string, string, string)	**EXPERIMENTAL:** Transcodes the string contents to the specified encoding. Refer to FAQ: Using Encode and Decode Functions.
encode	integer *encode*(string, string, string)	**EXPERIMENTAL:** Transcodes the source buffer to the string encoding. Refer to FAQ: Using Encode and Decode Functions.
formatib	integer *formatib*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in SAS IB. The first parameter is the value to format, the second is the W.D format, and the third is the formatted SAS IB number.
formatpd	integer *formatpd*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in SAS PD. The first parameter is the value to format, the second is the W.D format, and the third is the formatted SAS PD number.
formatpiccomp	integer *formatpiccomp*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in COMP. The first parameter is the value to format, the second is the PIC format, the third is the formatted COMP number.
formatpiccomp3	integer *formatpiccomp3*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in COMP-3. The first parameter is the value to format, the second is the PIC format, the third is the formatted COMP-3 number.
formatpiccomp5	integer *formatpiccomp5*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in COMP-5. The first parameter is the value to format, the second is the PIC format, and the third parameter is the formatted COMP-5 number.
formatpicsigndec	integer *formatpicsigndec*(real, string, string[, boolean[, boolean]])	**EXPERIMENTAL:** Returns a number formatted in COBOL signed decimal. The first parameter is the number to be formatted, the second is the PIC format, and the third will be the formatted COBOL Signed Decimal number. The fourth parameter will return true if there is an encoding, false is the default for ASCII. The fifth parameter will be true for trailing sign orientation, which is the default.

Function	Syntax	Description
formats370fib	integer *formats370fib*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in z/OS integer. The first parameter is the value to be formatted, the second is the W.D format, and the third is the formatted z/OS integer.
formats370fpd	integer *formats370fpd*(real, string, string)	**EXPERIMENTAL:** Returns a number formatted in z/OS packed decimal. The first parameter is the value to be formatted, the second is the W.D format, and the third is the formatted z/OS packed decimal.
ib	real *ib*(string[, string])	**EXPERIMENTAL:** Returns a number from a SAS IB value. The first parameter is the IB value, and the second is the W.D format.
normsinv	real *normsinv*(real)	**EXPERIMENTAL:** Returns the inverse of the cumulative standardized normal distribution.
parameter	string *parameter*(integer)	**EXPERIMENTAL:** Returns the value of a parameter, or null if the parameter does not exist.
parametercount	integer *parametercount*()	**EXPERIMENTAL:** Returns the number of parameters available.
pd	real *pd*(string[, string])	**EXPERIMENTAL:**Returns a number from a SAS PD value. The first string contains the number, the second contains the W.D format.
piccomp	real *piccomp*(string[, string])	**EXPERIMENTAL:** Returns a number from a COMP value. The first string contains the number, the second contains the PIC format.
piccomp3	real *piccomp3*(any[, string])	**EXPERIMENTAL:** Returns a number from a COMP-3 value. The first string contains the number, the second contains the PIC format.
piccomp5	real *piccomp5*(string[, string])	**EXPERIMENTAL:** Returns a number from a COMP-5 value. The first string contains the number, the second contains the PIC format.
picsigndec	real *picsigndec*(string[, string[, boolean[, boolean]]])	**EXPERIMENTAL:** Returns a number from a COBOL signed decimal value. The first string is the number, the second contains the PIC format, the third is false if ASCII encoded, and the fourth is true if using a trailing sign.
s370fib	real *s370fib*(string[, string])	**EXPERIMENTAL:** Returns a number from a z/OS integer. The first string is the number, the second is the W.D format.
s370fpd	real *s370fpd*(string[, string])	**EXPERIMENTAL:** Returns a number from a z/OS packed decimal. The first string is the number, the second is the W.D format.

For more information about experimental testing, refer to What is Experimental?

decode Function

Transcodes the string contents to the specified encoding.

Syntax

integer *decode*(string, string, string)

encode Function

Transcodes the source buffer to the string encoding.

Syntax

integer *encode*(string, string, string)

formatib Function

Returns a number formatted in SAS IB. The first parameter is the value to format, the second is the W.D format, and the third is the formatted SAS IB number.

Syntax

integer *formatib*(real, string, string)

formatpd Function

Returns a number formatted in SAS PD. The first parameter is the value to format, the second is the W.D format, and the third is the formatted SAS PD number.

Syntax

integer *formatpd*(real, string, string)

formatpiccomp Function

Returns a number formatted in COMP. The first parameter is the value to format, the second is the PIC format, the third is the formatted COMP number.

Syntax

integer *formatpiccomp*(real, string, string)

formatpiccomp3 Function

Returns a number formatted in COMP-3. The first parameter is the value to format, the second is the PIC format, the third is the formatted COMP-3 number.

Syntax

integer *formatpiccomp3*(real, string, string)

formatpiccomp5 Function

Returns a number formatted in COMP-5. The first parameter is the value to format, the second is the PIC format, and the third parameter is the formatted COMP-5 number.

Syntax

integer *formatpiccomp5*(real, string, string)

formatpicsigndec Function

Returns a number formatted in COBOL signed decimal. The first parameter is the number to be formatted, the second is the PIC format, and the third will be the formatted COBOL Signed Decimal number. The fourth parameter will return true if there is an encoding, false is the default for ASCII. The fifth parameter will be true for trailing sign orientation, which is the default.

Syntax

integer *formatpicsigndec*(real, string, string[, boolean[, boolean]])

formats370fib Function

Returns a number formatted in z/OS integer. The first parameter is the value to be formatted, the second is the W.D format, and the third is the formatted z/OS integer.

Syntax

integer *formats370fib*(real, string, string)

formats370fpd Function

Returns a number formatted in z/OS packed decimal. The first parameter is the value to be formatted, the second is the W.D format, and the third is the formatted z/OS packed decimal.

Syntax

integer *formats370fpd*(real, string, string)

ib Function

Returns a number from a SAS IB value. The first parameter is the IB value, and the second is the W.D format.

Syntax

real *ib*(string[, string])

normsinv Function

Returns the inverse of the cumulative standardized normal distribution.

Syntax

real *normsinv*(real)

Arguments

real

  is a real number to be passed to the function

parameter Function

Returns the value of a parameter, or null if the parameter does not exist.

Syntax

string *parameter*(integer)

Arguments

string

  contains the value returned by the function for the specified parameter

integer

  is the number of the parameter whose value is to be returned

Details

The *parameter* function returns <null> if the parameter does not exist.

parametercount Function

Returns the number of parameters available.

Syntax

integer *parametercount*()

Arguments

integer

  is the number of the parameters returned by the function

pd Function

Returns a number from a SAS PD value. The first string contains the number, the second contains the W.D format.

Syntax

real *pd*(string[, string])

Arguments

real

is the numeric value returned by the function

string

the first string is the unformatted number; the second (optional) string is the SAS format, specified as W.D (width.decimal)

string

[optional] the second string is the SAS format, specified as W(idth).D(ecimal)

Examples

```
// Declare a STRING variable to contain the packed decimal string
string pdstring
// Declare a REAL varialble to contain the real value of the packed decimal
real pdreal
// Declare a hidden INTEGER value to contain the packed decimal string
integer len
// Use FORMATPD function to create a packed decimal string
len = formatpd (30.56789, "8.2", pdstring);
// Format the packed decimal number into a real number of length 8 with 2
decimal places
pdreal = pd(pdstring, "8.2");
```

Results: This should return with a value of 30.57

piccomp Function

Returns a number from a COMP value. The first string contains the number, the second contains the PIC format.

Syntax

real *piccomp*(string[, string])

piccomp3 Function

Returns a number from a COMP-3 value. The first string contains the number, the second contains the PIC format.

Syntax

real *piccomp3*(any[, string])

piccomp5 Function

Returns a number from a COMP-5 value. The first string contains the number, the second contains the PIC format.

Syntax

real *piccomp5*(string[, string])

picsigndec Function

Returns a number from a COBOL signed decimal value. The first string is the number, the second contains the PIC format, the third is false if ASCII encoded, and the fourth is true if using a trailing sign.

Syntax

real *picsigndec*(string[, string[, boolean[, boolean]]])

s370fib Function

Returns a number from a z/OS integer. The first string is the number, the second is the W.D format.

Syntax

real = *s370fib*(string, format_str)

Arguments

string

    the octet array containing IBM mainframe binary data to convert

format_str

    the string containing the w.d. format of the data

s370fpd Function

Returns a number from a z/OS packed decimal. The first string is the number, the second is the W.D format.

Syntax

real *s370fpd*(string[, string])

Data Job Expressions Node

The DataFlux® Data Management Studio Expressions node is a utility that allows 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*.

Technical Support

- Frequently Asked Questions

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
- Selecting Output Fields
- Sub-Setting
- Initializing and Declaring Variables
- Saving Expressions
- Counting Records
- Debugging and Printing Error Messages

The EEL allows you to format and alter data through built-in functions and external processes. Specifically, you can use the following to structure and manipulate data:

- Creating Groups
- Retrieving and Converting Binary Data
- Supporting COBOL
- Using Array Functions
- Using Blue Fusion Functions
- Using Date and Time Functions
- Using Database Functions
- Using Encode and Decode Functions
- Using File Functions
- Using Integer and Real Functions
- Using Regular Expressions Functions
- Using String Functions

FAQ: Testing and Evaluating

In order to test an expression prior to 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, 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, **Generate rows when no parent is specified** must not be selected.

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.

FAQ: Selecting Output Fields

Some fields are used for calculation or to contain intermediate values, but are not meaningful in the output. As you are testing or building scripts, there may be a need to exclude fields from the output.

Exercise: How do I exclude some fields in the expression from being listed in the output?

You accomplish this by using the *hidden* keyword 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 string field, *noDisplay*, is not output to the **Preview** tab.

Verify this by removing the parameter *hidden* from the string noDisplay declaration. Observe that *noDisplay* is now output.

FAQ: Sub-Setting

In working with large record sets in the Data Job Editor, testing new jobs can be time consuming. Shorten the time to build your expression by testing 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 employed 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.

```
// 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
```

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.

FAQ: Initializing and Declaring Variables

As an expression is being evaluated, each row updates with the values of the fields in the expression. This may lead to re-initialization of certain variables in the expression. You may 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 only once 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.

FAQ: Saving 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 may load using **Load**.

FAQ: Counting Records

Exercise 1: How do I count the number of records in a table using Expression Engine Language (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 on 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, you would 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 getting updated after each row iteration.

Note: Field names must be enclosed in grave accents (ASCII &#96;) rather than apostrophes (ASCII &#39;).

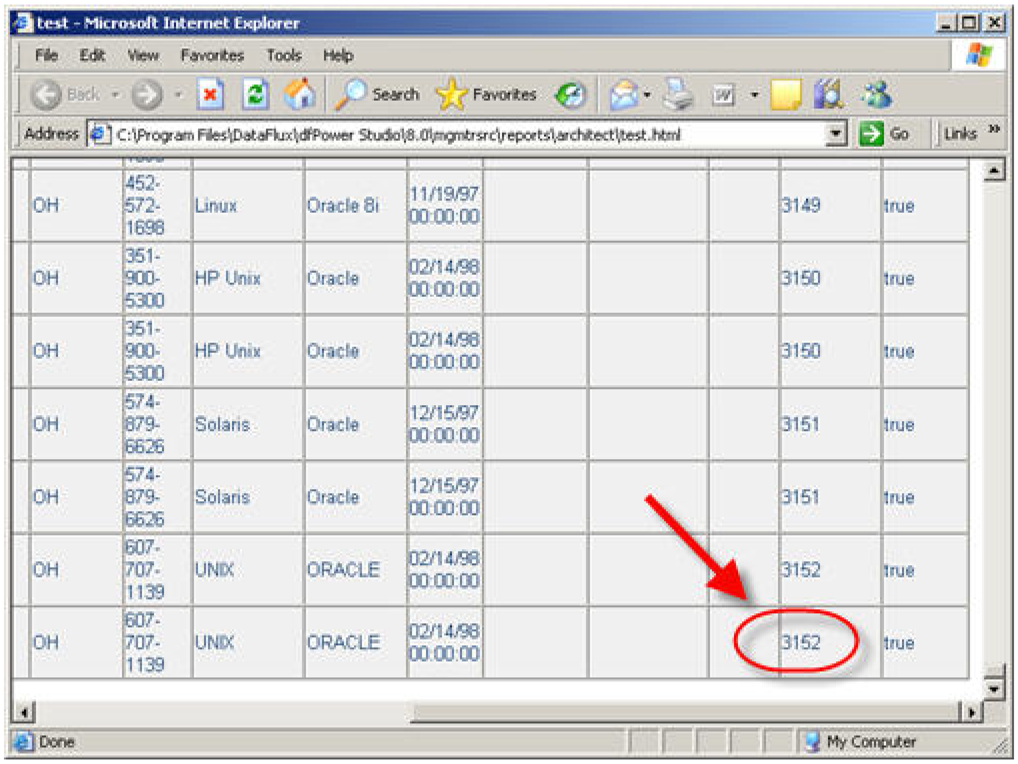
Exercise 2: How do I see the final count of the records in a table instead of seeing it get incremented by one on every row?

Declare a *count* variable as *hidden*, and 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 run it, the very last row should display the total number of records in the table that are not null.



test - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Program Files\DataFlux\dfPower Studio\8.0\mgmtrsrc\reports\architect\test.html
Go Links
Done
My Computer

OH	452-572-1698	Linux	Oracle 8i	11/19/97 00:00:00				3149	true
OH	351-900-5300	HP Unix	Oracle	02/14/98 00:00:00				3150	true
OH	351-900-5300	HP Unix	Oracle	02/14/98 00:00:00				3150	true
OH	574-879-6626	Solaris	Oracle	12/15/97 00:00:00				3151	true
OH	574-879-6626	Solaris	Oracle	12/15/97 00:00:00				3151	true
OH	607-707-1139	UNIX	ORACLE	02/14/98 00:00:00				3152	true
OH	607-707-1139	UNIX	ORACLE	02/14/98 00:00:00				3152	true

Displaying the Final Record Count

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 being displayed. Therefore, you can filter all the other rows from your view except the pushed one.

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()
```

FAQ: Debugging and Printing Error Messages

Exercise: Is there a way to print error messages or to get debugging information?

You may 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 we could output the final count to the statistics file. In the Post-processing section you would have:

```
// 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)
```

FAQ: Creating Groups

Expressions provide the ability to organize content into groups. The Expression Engine Language (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.

> **Note:** 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 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** and double-click the **Standardization** node.
2. In the **Standardization Properties** dialog, select **State** and specify the **State/Province (Abbreviation)** Definition. This creates a new field called STATE_Stnd.
3. Click **Additional Outputs** and select all.
4. Click **OK**.
5. In the **Standardization Properties** dialog, click **OK**.
6. In the Expression Properties dialog, click **Grouping**. Make sure that Grouping is now by STATE_Stnd and not STATE.
7. 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.

Step 1: Connect to a Data Source

1. Connect to the Purchase table in the DataFlux sample database.
2. In the **Data Source Properties** dialog, click **Add All**.
3. Find the Field Name for ITEM AMOUNT. Change the Output Name to ITEM_AMOUNT.

Step 2: Sorting 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.

1. In the Data Job Editor, click **Utilities** and double-click **Data Sorting.** This adds a Data Sorting node.
2. In the **Data Sorting Properties** dialog, select DEPARTMENT and set the **Sort Order** to Ascending.
3. Click **OK**.

Step 3: Creating Groups

To create groups out of the incoming data, add another Expression node to the job after the sorting step.

1. In the Expression Properties dialog, click **Grouping**. The following three tabs are displayed: Group Fields, Group Pre-expression, and Group Post-expression.
2. On the Group Fields tab, select DEPARTMENT.
3. 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
```

4. 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
...	...	...	...

FAQ: Retrieving and Converting Binary Data

Dataflux® expressions provide 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.

Big-Endian and Little-Endian Format

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 one and eight, inclusive, with the default being four. The optional decimal (d) must be between zero and 10, inclusive.

The w.d formats/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 has been 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 being 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 one and eight, inclusive, with the default being four. 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.

FAQ: 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.

- Reading binary data
- Formatting binary data

Reading

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 (two, four, or eight) to consume by comparing the sum of the nines in the integer and fraction portions to fixed ranges. If the sum is less than five, two bytes are consumed. If the sum is greater than four and less than 10, four bytes are consumed. If the sum is greater than nine and less than 19, eight 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), four 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 one. If the new value is odd, one is added to make it even. The result is then divided by two. As such, S9(7) would mean four bytes to consume. The packed data will always be 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 non-zero indicates the string is EBCDIC. The default ebcdic setting is false.
- **boolean trailing** - The boolean when set to non-zero 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 nines 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 nines 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 non-zero indicates to format in EBCDIC.
- **boolean trailing** - The boolean when non-zero 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.

FAQ: Using Array Functions

In this section, you will find additional information about arrays, including:

- Creating an Array
- Retrieving Elements from an Array
- Changing an Array Size
- Determining an Array's Size
- Finding Common Values Between Columns Using Arrays

Creating 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

Exercise: How do I retrieve elements from an array?

To retrieve elements from an array, use the following example, which 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

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

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

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:

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 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 may 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_co
mmun)
                                         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
```

FAQ: Using Blue Fusion Functions

Once a Blue Fusion object is defined and initialized, the function methods listed can be used within the Expression node.

Exercises

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 Quality Knowledge Base (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, type the following expressions:

```
// Expression

// define mc as the return string that contains the matchcode
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 matchcode for the string Washington D.C.,
// using the City definition at a sensitivity of 85, and
// put the result in mc
ret = bf.matchcode("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, after you initialize the Blue Fusion object in the Pre-processing tab, type the following expressions:

```
// 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, type 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, type 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, type 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 proper case
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, type 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'
```

FAQ: Using Date and Time Functions

In this section, you will find additional information about date and time functions, including:

- Finding Today's Date
- Formatting a Date
- Extracting Parts from a Date
- Adding or Subtracting from a Date
- Comparing Dates

Finding 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

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 Expression Engine Language (EEL), use the *formatdate*() function:

```
string formatdate(date, string)
```

The *formatdate*() functions 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

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

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
```

FAQ: Using Database Functions

Connecting 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 if there are any names that match the names 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
```

FAQ: 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 and decode functions. 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.

FAQ: Using File Functions

File Operations

Exercise 1: How do I open a file?

To open a file in Expression Engine Language (EEL), use this expression:

```
// Expression
File f
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 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 have selected **Generate rows when no parent is specified**. The file cursor advances one line in the text file for the 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 at the completion of the loop, the input string has a null value.

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()
```

⚠ **Caution:** 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. As an example, in order to append to the end of a file you would select **Generate rows when no parent is specified**, and type:

```
// 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 check if 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 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 have 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 will be 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?

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 employing 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.

FAQ: Using Integer and Real Functions

Integers and real types are basic data types in Expression Engine Language (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 if the variable has a numeric value?

The *isnumber*() built-in function can be used to determine if 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

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 may 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 be interpreted as false. Any non-zero 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 may be important.

Exercise: What is the range/precision for real and integer values?

Integer types are stored as 64-bit signed quantities with a range of -2 x 10^63 to 2 x 10^63 - 1.

Real types are stored as high precision values with an approximate precision of 44 digits and a range of 5.0 x 10^-324 to 1.7 x 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)
```

FAQ: 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 if my regex pattern matches part of my input?

To see if your regex pattern finds a match in the input string, use the following 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 you want to match, use the following 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."

FAQ: Using String Functions

Determining Type

The following exercises demonstrate how to determine the data type of a string.

Exercise 1: How do I determine if an entry is a string?

To determine if the string is a *string* type, use the *typeof*() function:

```
string typeof(any)
```

The *typeof*() function returns the type of data 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 if a string is made up of alphabetic characters?

To determine if 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 employ 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 suited for arrays, while *parse*() is useful for returning individual strings.

ASCII Conversions

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.

String Manipulations

Frequently, when working with strings, you may 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 white-space 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)
```

For example:

```
// Expression
string changeCase
changeCase="MixedCase"
string newCase
newCase=upper(changeCase)
```

Comparing and Matching

EEL lets you compare strings, find differences between strings, and search for substrings.

Exercise 1: How do I compare two strings?

To compare two strings, use an equal comparison (==). For example:

```
// Expression
string test
boolean match

// initialize
test="Hello"
match=false

// compare string values
if(test=="Hello") then
     match=true
```

To get a more in-depth comparison, consider the *compare*() and *edit_distance*() functions.

The *compare*() function compares two strings:

```
integer compare(string, string, boolean)
```

It returns:

- -1 if first < second
- 0 if first equals second
- 1 if first > second

If the third parameter is true, the comparison is case insensitive. The comparison between two strings is done lexicographically.

Another similar function is *edit_distance*():

```
integer edit_distance(string, 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 if the first string matches the second string, which may 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**). Questions marks may be used as a wildcard, but will only be matched 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.

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
```

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 zero, all occurrences will be 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 if 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);
```

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 if the string contains control characters. For a list of control characters, see Appendix A: ASCII Control Characters.

Evaluating Strings

EEL allows 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 though 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.

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" & 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
```

Appendixes

The following topics are available in the Appendix section of the DataFlux Expression Language Reference Guide.

- Appendix A: Reserved Words
- Appendix B: ASCII Values
- Appendix C: Encoding

Appendix A: Reserved Words

The following list of reserved words cannot be used for label names:

and	global	public
array	goto	real
begin	hidden	return
boolean	if	static
bytes	integer	step
call	not	string
date	null	then
else	or	to
end	pointer	visible
for	private	while

Appendix B: ASCII Values

The following tables contain the ASCII printable and control characters that can be represented by decimal values.

ASCII Printable Characters

The following table contains the ASCII printable characters that can be represented by decimal values:

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
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
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 C: 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	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	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	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	An 8-bit character encoding covering the modern Greek language along with mathematical symbols derived from Greek.
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. May 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.
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 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 pagecharacter 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.

Glossary

A

ASCII

American Standard Code for Information Interchange

C

Comments

Comments are text within a code segment that are 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).

E

EEL

Expression Engine Language

EOF

end of file

I

IEEE 754

The IEEE Standard for Binary Floating-Point Arithmetic. This standard applies to how floating point numbers are represented and the operations related to them.

Q

QKB

The QKB, or Quality Knowledge Base, is a collection of files and configuration settings that contain all DataFlux data management algorithms. The Quality Knowledge Base is directly editable using dfPower Studio.

Quality Knowledge Base

The collection of files and configuration settings that contain all DataFlux data management algorithms. The Quality Knowledge Base is directly editable using dfPower Studio.

R

regular expression

A mini-language composed of symbols and operators that enables you to express how a computer application should search for a specified pattern in text. A pattern may then be replaced with another pattern, also described using the regular expression language.

Index

A

abs, 80

aparse, 100, 101, 172

array, 7, 18, 150

  changing size, 151

  creating, 150

  determining size, 151

  dim, 18

  example, 150

  exercise, 150

  FAQ, 150

  finding values, 152

  get, 19

  retrieving, 151

  set, 20

asc, 100, 102, 173

ASCII, 181

  conversions, 173

assignment, 6

B

begin/end, 10

big-endian, 143

binary data, 143

  converting, 145

  FAQ, 143

  IBM mainframe, 146

  retrieving, 143

Blue Fusion, 21, 154

  bluefusion_initialize, 26

  case, 22

  FAQ, 154

  gender, 23

  getlasterror, 24

  identify, 25

  initialize, 26

  loadqkb, 27

  matchcode, 28

  pattern, 30

  standardize, 31, 155

bluefusion_initialize, 21, 26

boolean, 13, 32

  seteof, 42

boolean ebcdic, 148

boolean trailing, 148

C

call, 10

case, 21, 22

category

string, 170

Category

array, 18, 19, 20

Blue Fusion, 21, 26

boolean, 32, 42

data input, 37, 38, 39, 40, 43

database, 32, 36, 37, 160

date and time, 44, 46

encode and decode, 127

event, 47

execution, 48, 49, 50

external file, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64

file, 50

information/conversion, 66, 67, 68, 69, 71

logging, 73, 74, 75

lookup/reference, 110

macro variable, 76, 78, 79

mathematical, 80, 81, 82, 84, 85

regular expression, 86, 88, 89, 90, 91, 93, 94, 96, 98

search, 99

string, 100, 106

ceil, 80, 81

chr, 100, 103, 173

close, 50, 52

COBOL, 147

Formating, 149

coercion, 15

compare, 100, 104, 175

compile, 86

control characters, 182

copyfile, 50, 53

counting records, 138

FAQ, 138

D

data input, 37

fieldcount, 38

fieldname, 38

fieldtype, 39

fieldvalue, 40

setfieldvalue, 43

Data Job Editor, 133

database, 32, 160

connecting, 160

dbconnect, 36

dbdatasources, 37

listing, 161

date, 13, 44

date and time, 44, 158

adding, 159

comparing, 160

extracting, 159

finding, 158

formatdate, 44, 159
formatting, 158
subtracting, 159
today, 46
dbconnect, 32, 36, 160
dbdatasources, 32, 37, 161
debug, 140
declaration, 2, 4
decode, 127, 162
deletefile, 50, 53
dim, 18

E

edit_distance, 100, 105, 175
encode, 127, 162
encode and decode
decode, 127
encode, 127
encoding, 162, 182
endian, 147
error messages
debug, 140
print, 140
event, 47
raiseevent, 47
execute, 50, 54
execution, 48
pushrow, 48
setoutputslot, 49
sleep, 50
expression
boolean, 13
coercion, 15
date, 13
integer, 12
mathematical, 14
null, 14
real, 12
string, 12
Expression Language
overview, 2
expressions, 10
Expressions Node, 133
external file
close, 52
copyfile, 53
deletefile, 53
execute, 54
filedate, 55
fileexists, 56
open, 57
position, 58
readbytes, 59
readline, 60
seekbegin, 61

seekcurrent, 62

seekend, 63

writebytes, 64

F

FAQ, 134, 135, 136, 138, 140, 141, 143, 147, 154, 158, 160, 162, 167, 170

array, 150

Blue Fusion, 154

COBOL, 147

counting records, 138

database, 160

date and time, 158

debug, 140

decode, 162

encode, 162

evaluate, 135

expressions, 138

file, 162

groups, 141

integer, 167

output fields, 136

print, 140

real, 167

string, 170

sub-setting, 136

test, 135

variable, 138

fieldcount, 37, 38

fieldname, 37, 38

fieldtype, 37, 39

fieldvalue, 37, 40

file, 162

copying, 165, 166

deleting, 166

manipulating, 165

opening, 162

renaming, 166

writing, 164

file operations, 162

filedate, 50, 55

fileexists, 56

findfirst, 86, 88

findnext, 86, 89

floor, 80, 82

for, 9

format

big-endian, 143

little-endian, 143

format_str, 145

formatdate, 44, 159

formatib, 127

formatpd, 127

formatpiccomp, 127

formatpiccomp3, 127

formatpiccomp5, 128
formatpicsigndec, 128
formats370fpd, 128
function, 15
  abs, 80
  aparse, 101, 172
  asc, 102, 173
  bluefusion, 154
  bluefusion_initialize, 21, 26
  case, 21, 22
  ceil, 81
  chr, 103, 173
  close, 50, 52
  compare, 104, 175
  compile, 86
  copyfile, 50, 53
  dbconnect, 32, 36
  dbdatasources, 32, 37
  decode, 127, 162
  deletefile, 50, 53
  dim, 18
  edit_distance, 105, 175
  encode, 127, 162
  execute, 50, 54
  fieldcount, 37, 38
  fieldname, 37, 38
  fieldtype, 37, 39
  fieldvalue, 37, 40
  file, 162
  filedate, 50, 55
  fileexists, 50, 56
  findfirst, 88
  findnext, 89
  floor, 82
  formatdate, 44, 159
  formatib, 127
  formatpd, 127
  formatpiccomp, 127
  formatpiccomp3, 127
  formatpiccomp5, 128
  formatpicsigndec, 128
  formats370fpd, 128
  gender, 21, 23
  get, 18, 19
  getlasterror, 21, 24
  getvar, 78
  has_control_chars, 106, 178
  ib, 128
  identify, 21, 25
  initialize, 26
  inlist, 99
  instr, 107, 176
  integer, 167
  isalpha, 66, 171

isblank, 67

isnull, 68, 171

isnumber, 69

left, 107, 171

len, 108, 174

loadqkb, 21, 27

locale, 70

logmessage, 73

lower, 109, 174

match_string, 110, 176

matchcode, 21, 28

matchlength, 90

matchstart, 91

max, 82

mid, 111, 171

min, 83

mkdir, 112

movefile, 50, 56

normsinv, 129

open, 50, 57

parameter, 129

parametercount, 129

parse, 113, 173

pattern, 21, 30, 114, 177

pd, 130

piccomp, 130

piccomp3, 131

piccomp5, 131

picsigndec, 131

position, 50, 58

pow, 84

print, 32, 74

pushrow, 48, 135, 136

raiseerror, 74

raiseevent, 47

readbytes, 50, 59

readline, 50, 60

readrow, 32, 41

real, 167

regular expression, 169

replace, 93, 114, 177

right, 116, 171

rmdir, 116

round, 85

rowestimate, 32, 42, 136

s370fib, 131

s370fpd, 132

seekbegin, 50, 61

seekcurrent, 50, 62

seekend, 50, 63

seekwritebytes, 50

sendnodestatus, 75

set, 18, 20

seteof, 32, 42, 136

setfieldvalue, 37, 43
setoutputslot, 48, 49
setvar, 79
sleep, 48, 50
sort, 117
sort_words, 118
standardize, 21, 31
string, 170
substringcount, 94
substringlength, 96
substringstart, 98
toboolean, 71
todate, 119
today, 44, 46
tointeger, 120
toreal, 121
tostring, 121
trim, 122, 174
typeof, 71, 170
upper, 123, 174
username, 123
vareval, 124, 178
varset, 79
writebytes, 64
writeline, 50, 65

G

gender, 21, 23
gender analysis, 156
get, 19
getlasterror, 21, 24
getvar, 76, 78
  using, 77
global, 5
goto, 6
groups, 141
  creating, 141

H

has_control_chars, 100, 106, 178

I

ib, 128
IBM mainframe, 146
identify, 21, 25
if/else, 8
information/conversion, 66
  isalpha, 66
  isblank, 67
  isnull, 68
  isnumber, 69
  toboolean, 71
initialize, 26
inlist, 99
instr, 107, 176
integer, 12, 145

casting, 167
operations, 168
precision, 168
range, 168
rounding, 169
isalpha, 66, 171
isblank, 66, 67
isnull, 66, 68, 171
isnumber, 66, 69

L

label, 3, 6
label names, 180
large records, 136
left, 100, 107, 171
len, 108, 174
little-endian, 143
loadqkb, 21, 27
locale, 70
logging, 73
logmessage, 73
raiseerror, 74
sendnodestatus, 75
logmessage, 73
lookup/reference
match_string, 110
lower, 109, 174

M

macro variable, 76
getvar, 78
setvar, 79
using, 76
varset, 79
match_string, 110, 176
matchcode, 21, 28
creating, 155
matchlength, 86, 90
matchstart, 86, 91
mathematical, 14, 80
abs, 80
ceil, 81
floor, 82
pow, 84
round, 85
max, 82
method
close, 52
open, 57
position, 58
readbytes, 59
readline, 60
seekbegin, 61
mid, 111, 171
min, 83

mkdir, 112

movefile, 50, 56

N

native endian, 147

normsinv, 129

null, 14

O

object, 16

  Blue Fusion, 154

  close, 52

  dbconnect, 32

  open, 57

  position, 58

  readbytes, 59

  readline, 60

  seekbegin, 61

open, 50, 57

operations, 168

operators, 10

overview, 2

P

parameter, 129

parametercount, 129

parse, 100, 113, 173

pattern, 21, 30, 100, 114, 177

pattern analysis, 157

pd, 130

piccomp, 130

piccomp3, 131

piccomp5, 131

picsigndec, 131

position, 50, 58

pow, 80, 84

print, 32, 74, 140

printable characters, 181

public, 5

pushrow, 48, 135, 136

Q

QKB, 154

Quality Knowledge Base, 154

questions, 134

R

raiseerror, 73, 74

raiseevent, 47

readbytes, 50, 59

readline, 50, 60

readrow, 32, 41

real, 12, 145

  casting, 167

  operations, 168

  precision, 168

  range, 168

rounding, 169

records

large, 136

regular expression, 86, 169

compile, 86, 169

findfirst, 88, 169

findnext, 89, 169

matchlength, 90, 169

matchstart, 91, 169

replace, 93

substringcount, 94

substringlength, 96

substringstart, 98

replace, 86, 93, 114, 177

reserved words, 180

return, 7

right, 100, 116, 171

rmdir, 116

round, 80, 85

rounding, 169

rowestimate, 32, 42, 136

S

s370fib, 131

s370fpd, 132

search, 99

inlist, 99

seekbegin, 50, 61

seekcurrent, 50, 62

seekend, 50, 63

sendnodestatus, 73, 75

set, 20

seteof, 32, 42, 136

setfieldvalue, 37, 43

setoutputslot, 48, 49

setvar, 76, 79

using, 77

sleep, 48, 50

sort, 117

sort_words, 118

standardize, 21, 31

statement, 3, 6

begin/end, 10

call, 10

for, 9

goto, 6

if/else, 8

label, 6

return, 7

while, 9

string, 100, 145, 170

aparse, 172

asc, 173

chr, 173

compare, 175

control characters, 178
edit_distance, 175
evaluate, 178
has_control_chars, 106
instr, 176
isalpha, 171
isnull, 171
left, 171
len, 174
lower, 174
manipulation, 173
match, 175
match_string, 176
mid, 171
parse, 173
parsing, 172
pattern, 177
replace, 177
right, 171
substrings, 171
trim, 174
type, 170
typeof, 170
upper, 174
vareval, 178
string buffer, 148
string format_str, 148
strings, 12
substringcount, 86, 94
substringlength, 86, 96
substringstart, 86, 98
symbols, 4
global, 5
public, 5

T

time, 44
toboolean, 66, 71
todate, 119
today, 44, 46
tointeger, 120
toreal, 121
tostring, 121
trim, 100, 122, 174
typeof, 71, 170

U

upper, 123, 174
username, 123

V

vareval, 100, 124, 178
variable, 138
declaring, 138
initializing, 138
varset, 76, 79

W

while, 9

writebytes, 50, 64

writeline, 50, 65