IBM Cognos Data Manager
Version 10.2.1

Function and Scripting Reference Guide

Note
Before using this information and the product it supports, read the information in "Notices" on page 107.

Product Information

This document applies to IBM Cognos Business Intelligence Version 10.2.1 and may also apply to subsequent releases.

Contents

Introduction

This document is intended for use with the IBM® Cognos® Data Manager functions and the scripting language that you can use within the Cognos Data Manager engine or from Cognos Data Manager Designer.

The examples in this document use BackusNaur Form (BNF) to describe the syntax of the Data Manager language. The BNF syntax uses the conventions described in the following table.

Table 1. Conventions used for BNF syntax

Syntax	Description
A set of terminal symbols that are the words, commands, or punctuation of the language and command-line interface.	
A set of non-terminal symbols that are essentially place holders.	The symbols appear in angled brackets < >, for example, <refdata_file>. A definition of each non-terminal symbol may appear elsewhere in the syntax definition. However, not all non-terminal symbols are defined. For a complete definition of the scripting language, including the description of all non-terminal symbols, see the *IBM Cognos Function and Scripting Reference Guide*.
A set of rules that you apply when interpreting the BNF definitions:	• Colon colon equal sign (::=) means 'is defined to be'. • Square brackets [ ] indicate that the enclosed symbols are optional. • Braces { } indicate that the enclosed symbols may be repeated zero or more times. • The pipe symbol | indicates that you should choose only one of the items that it separates.

The following example defines the <options> symbol to be an optional -C, followed by a <var_list> symbol. It then defines the <var_list> symbol to be zero, one, or more instances of -V followed by a <name>=<value> pair:

```
<options> ::=
[-C] <var_list>

<var_list> ::=
{-V<name>=<value>
```

Audience

You should be familiar with Microsoft Windows and SQL. You should also have an understanding of multi-dimensional data analysis or Business Intelligence.

Finding information

To find IBM Cognos product documentation on the web, including all translated documentation, access one of the IBM Cognos Information Centers

(http://pic.dhe.ibm.com/infocenter/cogic/v1r0m0/index.jsp). Release Notes are published directly to Information Centers, and include links to the latest technotes and APARs.

You can also read PDF versions of the product release notes and installation guides directly from IBM Cognos product disks.

Forward-looking statements

This documentation describes the current functionality of the product. References to items that are not currently available may be included. No implication of any future availability should be inferred. Any such references are not a commitment, promise, or legal obligation to deliver any material, code, or functionality. The development, release, and timing of features or functionality remain at the sole discretion of IBM.

Accessibility features

IBM Cognos Data Manager does not currently support accessibility features that help users with a physical disability, such as restricted mobility or limited vision, to use this product.

IBM Cognos HTML documentation has accessibility features. PDF documents are supplemental and, as such, include no added accessibility features.

Samples disclaimer

The Sample Outdoors Company, Great Outdoors Company, GO Sales, any variation of the Sample Outdoors or Great Outdoors names, and Planning Sample depict fictitious business operations with sample data used to develop sample applications for IBM and IBM customers. These fictitious records include sample data for sales transactions, product distribution, finance, and human resources. Any resemblance to actual names, addresses, contact numbers, or transaction values is coincidental. Other sample files may contain fictional data manually or machine generated, factual data compiled from academic or public sources, or data used with permission of the copyright holder, for use as sample data to develop sample applications. Product names referenced may be the trademarks of their respective owners. Unauthorized duplication is prohibited.

Chapter 1. Function reference

IBM Cognos Data Manager provides predefined functions and operators that you can use, with the Cognos Data Manager scripting language, in derivations, derived dimensions, delivery output filters, variables, DataStream filters, and in JobStream procedure and condition nodes.

Functions may take zero, one, or more parameters and return a single value.

Conversion functions

Conversion functions convert data from one data type to another.

SetTimeZone

Sets a date or time to a date or time in a time zone.

The <value> input value can be of type CHAR, DATE or TIME. Returns a value of type DATE WITH TIME ZONE or TIME WITH TIME ZONE depending on the input value type.

For input values of type CHAR, you can specify the date or time format. If you omit a format, the input value must be in the default IBM Cognos Data Manager date format syyyy-mm-dd hh:mi:ss[.fffffffff] or time format hh:mi:ss[.fffffffff]

If the value is DATE WITH TIME ZONE or TIME WITH TIME ZONE already, NULL is returned.

To change the time zone, see “ToTimeZone” on page 10.

Syntax

```
SetTimeZone(<string>|<integer>, <value>[ , <format>])
```

Symbol	Description
<string>	The time zone. Possible values are as follows: • Local is the local time zone of the computer on which Data Manager is running. It is set from the operating system at startup not taking account of daylight saving. • UTC is Coordinated Universal Time (previously GMT). • GMT is Greenwich Mean Time (superseded by UTC). • ACDT is Australian Central Daylight Time (UTC+10:30). • ACST is Australian Central Standard Time (UTC+9:30). • AEDT is Australian Eastern Daylight Time (UTC+11:00). • AEST is Australian Eastern Standard Time (UTC+10:00). • AKDT is Alaska Daylight Time (UTC-8:00). • AKST is Alaska Standard Time (UTC-9:00). • ADT is Atlantic daylight Time (UTC-3:00). • AST is Atlantic Standard Time (UTC-4:00). • AWST is Australian Western Standard Time (UTC+8:00). • BST is British Summer Time (UTC+1:00). • CDT is Central Daylight Time (UTC-5:00). • CST is (US) Central Standard Time (UTC-6:00). • CEDT is Central European Daylight Time (UTC+2:00). • CEST is Central European Summer Time (UTC+2:00). • CET is Central European Time (UTC+1:00). • EDT is Eastern Daylight Time (UTC-4:00). • EST is (US) Eastern Standard Time (UTC-5:00). • EEDT is Eastern European Daylight Time (UTC+3:00). • EEST is Eastern European Summer Time (UTC+3:00). • EET is Eastern European Time (UTC+2:00). • JST is Japan Standard Time (UTC+9:00). • MDT is Mountain Daylight Time (UTC-6:00). • MST is (US) Mountain Standard Time (UTC-7:00). • NDT is Newfoundland Daylight Time (UTC-2:30). • NST is Newfoundland Standard Time (UTC-3:30). • NZDT is New Zealand Daylight Time (UTC+11:00). • NZST is New Zealand Standard Time (UTC+12:00). • NZT is New Zealand Time (UTC+12:00). • NFT is (Australian) Norfolk Island Time (UTC+11:30). • PDT is Pacific Daylight Time (UTC-7:00). • PST is (US) Pacific Standard Time (UTC-8:00).

Symbol	Description
	• SST is Singapore Standard Time (UTC+8:00). • WEDT is Western European Daylight Time (UTC+1:00). • WEST is Western European Summer Time (UTC+1:00). • WET is Western European Time (UTC). • WST is (Australian) Western Standard Time (UTC+8:00). • +/-hh:mi is the displacement from UTC, for example, -8:00, 07:43.
<integer>	The displacement from UTC in minutes. The value can be -1200 to -1300 (to allow for daylight saving)
<value>	A DATE or TIME, or a CHAR representation of a date or time value
<format>	The format of <value> if this is a string

Examples

- `SetTimeZone('local','121314','hhmiss')`
  This example returns the time value 12:13:14 +00:00 in the United Kingdom.
- `SetTimeZone ('local', '2006-12-31 12:13:14')`
  This example returns 2006-12-31 12:13:14 -5:00 in Ottawa, Canada.
- `SetTimeZone('PST','2006-12-31 12:13:14')`
  This example returns the time value '2006-12-31 12:13:14 -8:00'.
- `SetTimeZone('UTC','12:13:14 -5:00', 'hh:mi:ss stzh:tzm')`
  This example returns NULL because the time is already in a time zone.

ToChar

Returns the string representation of a value.

If the date format is used, then <value> must be a string value in the default IBM Cognos Data Manager date format of syyyy-mm-dd [hh:mi:ss[,fffffffff]]. The time part is optional as is fractions of a second within time and fractions of a second can be less than the maximum precision of 9.

ToChar also accepts string values and returns them without change.

Syntax

```
ToChar(<value> [ , <date format>])
```

Symbol	Description
<value>	A value of any data type
<date format>	The date format to use (only valid if <value> is a string)

Examples

These examples assume mynum equals 39 and mydate equals 2006-12-01.

- `ToChar(mynum)`
  This example returns '39'.
- `ToChar(mydate,'dd/mm/yy')`
  This example returns '01/12/06'.
- `ToChar(1=2)`
  This example returns 'FALSE'.

ToDate

Converts a string (of optional format) to a date or a date time.

The input value can be of type CHAR, DATE, or DATE WITH TIME ZONE. Returns a value of type DATE.

For input values of type CHAR, you can specify the time format. If you omit a format, the input value must be in the default IBM Cognos Data Manager time format which is yyyy-mm-dd [hh:mi:ss[.fffffffff] [stzh:tzm]]. The fractions of a second part is optional and can be less than the maximum precision of 9. The time zone part is also optional.

Input values of type DATE WITH TIME ZONE are converted to DATE, that is, the time zone part is dropped.

ToDate returns values of type DATE unchanged.

Syntax

```
ToDate (<date>[ , <format>])
```

Symbol	Description
<date>	A DATE or DATE WITH TIME ZONE value or the text representation of a date or date time
<format>	The format of <date> if <date> is a string

Examples

- `ToDate('2006-06-22 121314','yyyy-mm-dd hhmiss')`
  This example returns the date value 2006-06-22 12:13:14.
- `ToDate('2006-06-22 12:13:14 -5.00','yyyy-mm-dd hh:mi:ss stzh:tzm')`
- `ToDate('2006-06-22 12:13:14 -5.00','yyyy-mm-dd hh:mi:ss stzh.tzm')`
  These examples return the date value 2006-06-22 12:13:14.

ToDouble

Converts a value to a double-precision, floating-point number.

For input values of type INTEGER and FLOAT, ToDouble returns a value equal to the input value.

For input values of type CHAR or CLOB, ToDouble returns the value that the text represents. Where the entire input string cannot be interpreted as a decimal number, ToDouble returns the value represented by the left most part up to (but not including) the first character that is neither a digit nor the decimal point.

For input values of type NUMBER with a precision of 17 or fewer significant figures, ToDouble returns a value equal to the input value. For values of type NUMBER with a precision of greater than 17 significant figures, ToDouble returns the input value rounded to 17 significant figures.

For input values of type DATE or DATE WITH TIME ZONE, ToDouble returns the corresponding Julian date value.

For input values of type BOOLEAN, ToDouble returns zero where the input value is FALSE, and a non-zero value otherwise.

For input values of type TIME or TIME WITH TIME ZONE, ToDouble returns seconds from midnight.

For input values of type INTERVAL DAY TO SECONDS, ToDouble returns seconds.

For input values of type INTERVAL YEAR TO MONTH, ToDouble returns months.

For input value of type BINARY or BLOB, To Double returns the equivalent number.

Syntax

```
ToDouble(<value> [, scale])
```

Symbol	Description
<value>	A value of any data type
<scale>	A number value that specifies the maximum number of digits that can follow the decimal point

Examples

- `ToDouble('123.45')`
  This example returns 123.45.
- `ToDouble('123.4567,3')`
  This example returns 123.456.
- `ToDouble('123..45')`
  This example returns 123.00.
- `ToDouble(ToDate('01-01-2000','mm-dd-yyyy'))`
  This example returns 2451545.
- `ToDouble(1=2)`
  This example returns zero.
- `ToDouble(ToInterval(84000))`
  This example returns 84000.0.
- `ToDouble(ToTime(83999))`

This example returns 83999.0.

- `ToDouble(ToIntervalYM (37))`

  This example returns 37.0.

ToHex

Converts a value to a hex string. NULL values can also be converted.

BLOBs and CLOBs are truncated if the result is a hex string which is more than the IBM Cognos Data Manager maximum of 8000 bytes.

Syntax

```
ToHex(<value>)
```

Symbol	Description
<value>	A value of any data type

Examples

```
ToHex('ABC')
```

This example returns 0x41424300.

ToInteger

Converts a value to a number of type INTEGER.

For input values of type INTEGER, ToInteger returns the input value unchanged.

For input values of type CHAR or CLOB, ToInteger returns the value that the text represents. Where the entire input string cannot be interpreted as a decimal number, ToInteger returns the value represented by the left most part up to (but not including) the first character that is neither a digit nor the decimal point.

For input values of type DOUBLE, ToInteger returns the input value truncated to the nearest integer.

For input values of type DATE or DATE WITH TIME ZONE, ToInteger returns the equivalent Julian date (the number of days since December 31 4713BC.)

For values of type BOOLEAN, ToInteger returns zero where the input value is False and a non-zero integer otherwise.

For input values of type TIME or TIME WITH TIME ZONE, ToInteger returns seconds from midnight.

For input values of type INTERVAL DAY TO SECONDS, ToInteger returns seconds.

For input values of type INTERVAL YEAR TO MONTH, ToInteger returns months.

For input values of type BINARY or BLOB, ToInteger returns the equivalent number.

Syntax

```
ToInteger(<value>)
```

Symbol	Description
<value>	A value of any data type except NUMBER

Examples

- `ToInteger('123')`
  `ToInteger(123.56)`
  These examples return 123
- `ToInteger(ToDate('01-01-2001','mm-dd-yyyy'))`
  This example returns 2451911

ToIntervalDS

Converts a string (of optional format) or a number (representing a number of seconds) to a day to second interval.

The input value can be of type CHAR, INTERVAL DAY TO SECOND or any numeric data type. Returns a value of type INTERVAL DAY TO SECOND.

For input values of type CHAR, you can specify the interval format. If you omit a format, the input value must be in the default IBM Cognos Data Manager day to second interval format of sddddddddd hh:mi:ss[.fffffffff]. The fractions of a second part is optional and can be less than the maximum precision of 9. The number of days can also be less than the maximum precision of 9.

ToIntervalDS returns value of type INTERVAL DAY TO SECOND unchanged.

Syntax

```
ToIntervalDS(<string>|<integer>[ ,format]))
```

Symbol	Description
<string>	An INTERVAL DAY TO SECOND value or the text representation of the interval
<integer>	A value of any numeric type
<format>	The format of <string> if it is a string

Examples

- `ToIntervalDS('1 121314123','sddd hhmissfff')`
  This example returns the interval value 1 12:13:14.123
- `ToIntervalDS(1)`
  This example returns the interval value 0 00:00:01
- `ToIntervalDS (259199)`
  This example returns the interval value 2 23:59:59
- `ToIntervalDS(259199.123)`

This example returns the interval value 2 23:59:59.123

- `ToIntervalDS (-259199)`

  This example returns the interval value -2 23:59:59

ToIntervalYM

Converts a string (of optional format) or a number (representing number of months) to a year to month interval.

The input value can be of type CHAR, INTERVAL YEAR TO MONTH or INTEGER. Returns a value of type INTERVAL YEAR TO MONTH.

For input values of type CHAR, you can specify the interval format. If you omit a format, the input value must be in the default IBM Cognos Data Manager year to month interval format of syyyyyyyyy-mm. The number of years can be less than the maximum precision of 9.

ToIntervalYM returns value of type INTERVAL YEAR TO MONTH unchanged.

Syntax

```
ToIntervalYM(<string>|<integer>[ ,format]))
```

Symbol	Description
<string>	An INTERVAL YEAR TO MONTH value or the text representation of the interval
<integer>	An INTEGER value
<format>	The format of <string>

Examples

- `ToIntervalYM('1 11','syyy mm')`

  This example returns the interval value 1-11

- `ToIntervalYM (40)`

  This example returns the interval value 3-04

- `ToIntervalYM (-40)`

  This example returns the interval value -3 04

ToNumber

Converts a value to a number with a specified precision and scale.

Syntax

```
ToNumber(<value>,<precision>,<scale>)
```

Symbol	Description
<value>	A value of any type
<precision>	A number value that specifies the maximum number of digits included in the number

Symbol	Description
<scale>	A number value that specifies the maximum number of digits that can follow the decimal point

Examples

- `ToNumber('1234.56', 6, 2)`
  This example returns 1234.56
- `ToNumber(1234, 6, 2)`
  This example returns 1234.0

ToTime

Converts a string (of optional format) or a number, representing seconds from midnight, to a time.

The input value can be of type CHAR, TIME, TIME WITH TIME ZONE or any numeric data type.

Returns a value of type TIME.

For input values of type CHAR, you can specify the time format. If you omit a format, the input value must be in the default IBM Cognos Data Manager time format hh:mi:ss[.fffffffff] [stzh:tzm]. Both the fractions of a second (which can be less than the maximum precision of 9) and the time zone are optional.

Numeric values must be in the range 0-86399.999999999

Input values of type TIME are returned unchanged. Input values of type TIME WITH TIME ZONE are converted to TIME, that is, the time zone part is dropped.

Syntax

```
ToTime(<time>|<number>[,<format>])
```

Symbol	Description
<time>	A TIME or TIME WITH TIME ZONE value or the text representation of the time
<number>	A value of any numeric data type
<format>	The format of <time> if <time> is a string

Examples

- `ToTime('121314','hhmiss')`
  This example returns 12:13:14
- `ToTime('12:13:14 -5:00', 'hh:mi:ss stzh:tzm')`
  This example returns the time value '12:13:14 '
- `ToTime(86399.999)`
  This example returns the time value 23:59:59.999

ToTimeZone

Converts a date with time zone or time with time zone to a date or time value in a different time zone.

The input value can be of type CHAR, DATE WITH TIME ZONE, TIME, or TIME WITH TIME ZONE. Returns a value of type DATE WITH TIME ZONE or TIME WITH TIME ZONE depending on the input value type.

For input values of type CHAR, you can specify the date or time format. If you omit a format, the input value must be in the default IBM Cognos Data Manager date with time zone format syyyy-mm-dd hh:mi:ss[.fffffffff] [stzh:tzm] or time with time zone format hh:mi:ss[.fffffffff] [stzh:tzm]

If no time zone is given for the input value then the local time zone is presumed, that is, the time zone of the computer on which Cognos Data Manager is running.

Syntax

```
ToTimeZone (<string>|<integer>,<value>[ ,<format>])
```

Symbol	Description
<string>	The time zone. Possible values are • Local is the local time zone of the computer on which Cognos Data Manager is running. It is set from the operating system at startup not taking account of daylight saving. • UTC is Coordinated Universal Time (previously GMT) • GMT is Greenwich Mean Time (superseded by UTC) • ACDT is Australian Central Daylight Time (UTC+10:30) • ACST is Australian Central Standard Time (UTC+9:30) • AEDT is Australian Eastern Daylight Time (UTC+11:00) • AEST is Australian Eastern Standard Time (UTC+10:00) • AKDT is Alaska Daylight Time (UTC-8:00) • AKST is Alaska Standard Time (UTC-9:00) • ADT is Atlantic daylight Time (UTC-3:00) • AST is Atlantic Standard Time (UTC-4:00) • AWST is Australian Western Standard Time (UTC+8:00) • BST is British Summer Time (UTC+1:00) • CDT is Central Daylight Time (UTC-5:00) • CST is (US) Central Standard Time (UTC-6:00) • CEDT is Central European Daylight Time (UTC+2:00)

Symbol	Description
	• CEST is Central European Summer Time (UTC+2:00) • CET is Central European Time (UTC+1:00) • EDT is Eastern Daylight Time (UTC-4:00) • EST is (US) Eastern Standard Time (UTC-5:00) • EEDT is Eastern European Daylight Time (UTC+3:00) • EEST is Eastern European Summer Time (UTC+3:00) • EET is Eastern European Time (UTC+2:00) • JST is Japan Standard Time (UTC+9:00) • MDT is Mountain Daylight Time (UTC-6:00) • MST is (US) Mountain Standard Time (UTC-7:00) • NDT is Newfoundland Daylight Time (UTC-2:30) • NST is Newfoundland Standard Time (UTC-3:30) • NZDT is New Zealand Daylight Time (UTC+11:00) • NZST is New Zealand Standard Time (UTC+12:00) • NZT is New Zealand Time (UTC+12:00) • NFT is (Australian) Norfolk Island Time (UTC+11:30) • PDT is Pacific Daylight Time (UTC-7:00) • PST is (US) Pacific Standard Time (UTC-8:00)
	• SST is Singapore Standard Time (UTC+8:00) • WEDT is Western European Daylight Time (UTC+1:00) • WEST is Western European Summer Time (UTC+1:00) • WET is Western European Time (UTC) • WST is (Australian) Western Standard Time (UTC+8:00) • +/-hh:mi is the displacement from UTC, for example, -8:00, 07:43
<integer>	The time zone as the displacement from UTC in minutes. The value can be -1200 to +1300 (allows for daylight saving)
<value>	A DATE, DATE WITH TIME ZONE, TIME, TIME WITH TIME ZONE, or CHAR value which is the text representation of a date or time
<format>	The format of <value> if it is a string

Examples

- `ToTimeZone('UTC','31/12/2006 121314 +3:10', 'dd/mm/yyyy hhmiss stzh:tzm')`

  This example returns the date/time value 2006-12-31 09:03:14 0:00
- `ToTimeZone('EST', '2006-12-31 22:13:14 -08:10')`

  This example returns the date/time value 2007-01-01 01:23:14 -5:00
- `ToTimeZone('8:00', SetTimeZone('121314 03:00', 'hhmiss stzh:tzm'))`

  This example returns the time value 17:13:14 8:00

Control functions

Control functions give some control over how IBM Cognos Data Manager executes fact builds, dimension builds, and JobStreams.

This category also provides functions for file operations.

In addition to the control functions provided, there is a set of SQL cursor functions that allow you to prepare an SQL statement for execution, open a cursor for the statement, collect data from it, and then close the cursor. The purpose of these functions is to enable multiple processing of rows and columns of data. For more information, see "SQL cursor functions" on page 35.

ArrayAddItem

Adds an element to an array.

For more information, see "ARRAY" on page 91

Syntax

```
ArrayAddItem(<array>, <value>)
```

Symbol	Description
<array>	A variable of type ARRAY
<value>	The value to add which can be any data type

Examples

```
ArrayAddItem($ArrayVar,'some text')
```

This example adds 'some text' to the end of the array.

ArrayClear

Deletes all elements from an array and returns zero.

Syntax

```
ArrayClear(<array>)
```

Symbol	Description
<array>	A variable of type ARRAY

Examples

```
ArrayClear(SampArray)
```

This example deletes all items from the array SampArray and returns zero.

ArrayDeleteItem

Deletes the element from an array at the specified index and returns the number of remaining elements.

This function returns NULL, but does nothing to the array, if the specified element does not exist.

Syntax

```
ArrayDeleteItem(<array>,<index>)
```

Symbol	Description
<array>	A value of type ARRAY
<index>	The ordinal position, starting from one, of the element to delete

Examples

```
ArrayDeleteItem(myArray,2)
```

If myArray contains {'one','two','three','four'}, then this example deletes the second element from myArray (so that myArray contains {'one','three','four'}, and returns 3 (the number of elements that remain in myArray).

ArrayItem

Returns the element value of an array at the specified index.

This function returns NULL if the specified element does not exist.

Syntax

```
ArrayItem(<array>,<index>)
```

Symbol	Description
<array>	A value of type ARRAY
<index>	The ordinal position, starting from one, of the required element

Examples

- `ArrayItem(myArray,2)`

  If myArray contains {'one','two','three','four'} this example returns 'two'
- `ArrayItem(myArray,5)`

  If myArray contains {'one','two','three','four'} this example returns NULL (because 5 is outside the array bounds)

ArrayModifyItem

Modifies an array element at the specified index to the specified value.

Syntax

```
ArrayModifyItem( <array>, <index>, <value>)
```

Symbol	Description
<array>	A value of type ARRAY

Symbol	Description
<index>	The ordinal position, starting from one, of the element to modify
<value>	The new value to use which can be any data type

Examples

```
ArrayModifyItem(myArray,2,'mytext')
```

If myArray contains {'one','two','three','four'}, then this example modifies the value of the second element from 'two' to 'mytext'.

ArraySearch

Searches for an element in an array.

Searches through an array looking for the value. Returns the index if it finds it, or NULL. Indicate TRUE or FALSE to say whether the array is sorted or not. If sorted is not specified, assumes unsorted.

For information on sorting, see "ArraySort" on page 15.

Syntax

```
ArraySearch(<array>,<value>[,<sorted>])
```

Symbol	Description
<array>	A value of type ARRAY
<value>	The value to search for
<sorted>	Specify TRUE or FALSE to indicate whether the array has been sorted

Examples

In the following examples, the array named $Array is initialized to four items:

```
$Array := 'D' & 'C' & 'B' & 'A';
```

- `$pos_unsorted := ArraySearch( $Array, 'B' ) );`
  This example returns 3 as B is in the third slot of the array.
- `ArraySort( $Array ); $pos_sorted := ArraySearch( $Array, 'B' ) );`
  This example returns 2 as in the sorted array, B is in the second slot of the array.

ArraySize

Returns the number of elements in an array.

Syntax

```
ArraySize(<array>)
```

Symbol	Description
<array>	A value of type ARRAY

Examples

```
$num_elements := ArraySize($example_array);
```

This example counts the number of elements in the array named example_array and assigns the result to the variable num_elements.

ArraySort

Sorts an array in a way suitable for ArraySearch. The collating sequence of the sort is defined using the character setting for the computer on which IBM Cognos Data Manager is running.

Syntax

Returns the number of items in the array.

```
ArraySort (<array>)
```

Symbol	Description
<array>	A value of type ARRAY

For more information, see "ArraySearch" on page 14.

Audit

Writes a message, with an audit group of USER, to the build audit trail and returns TRUE or FALSE to indicate success or failure.

Because IBM Cognos Data Manager maintains audit information in the current catalog, you cannot use this function in a file-based project.

Syntax

```
Audit(<item>, <value>)
```

Symbol	Description
<item>	A user-defined keyword to which the message relates, of type CHAR
<value>	The text of the audit message, of type CHAR

Examples

```
Audit('START', 'Build 345')
```

AuditTrail

Returns the first available message that corresponds to an audit run identifier, audit group, and audit item combination.

Syntax

You can use the optional fourth parameter to select between messages that have the same identifier, group, and item combination.

```
AuditTrail(<audit_id>, [<group>], <item>[, <message>])
```

Symbol	Description
<audit_id>	An integer that identifies a specific execution of a fact build or dimension build.
<group>	The type of audit information required. For a list of permitted values and descriptions, see "Group values."
<item>	The item for which you require information. The value of <group> determines which items are available. If more than one matching item exists, IBM Cognos Data Manager returns the first encountered. For a list of permitted values and descriptions, see "Item values" on page 17.
<message>	An optional, wildcard specification of the audit message. The available wildcards are • percent symbol (%) which represents any number of characters • underscore (_) which represents a single character To include a literal % or _, escape the character with a backslash.

Group values

Group	Description
ACQUIRE	Information about source data acquisition. For example, the number of data rows that each data source contributes.
TRANSFORM	Information about the transformation of data. For example, the number of data rows that the transformation engine receives from each acquisition module.
DELIVER	Information about data delivery. For example, the number of data rows delivered.
INTERNAL	Information about internal IBM Cognos Data Manager structures. For example, the size of the hash table.

Group	Description
TIMING	Timing information. For example, the build start and end times, and the elapsed time for the build.

Item values

ACQUIRE Group

Item	Description
ROWLIMITS	A string representation of two numbers, for example, '10 1000'. The first number specifies the data sample rate, the second specifies the maximum row limit.
ROWS	A string that gives the name of a data source together with the three numbers, for example, '[Sales] 1000 2000 1000'. The first number specifies the physical number of rows that the data source retrieved from the database. The second number specifies the logical number of rows created. This may differ from the physical number of rows if, for example, you map more than one data source item to a DataStream, or if you pivot data. The third number specifies the number of output DataStream rows to which the data source contributes.

TRANSFORM Group

Item	Description
READ	The number of data rows that the DataStream makes available to the transformation engine.
ACCEPTED	The number of data rows that IBM Cognos Data Manager accepts.
REJECTED	The number of data rows that Cognos Data Manager rejects.
DIRECT	The number of candidate output data rows that come directly from the source data.
SUMMARY	The number of candidate output data rows that come from consolidation of the source data.
TOTAL	The total number of candidate output data rows.

DELIVER Group

Item	Description
NAME	Specifies an identifier, a name, and the delivery module used for a delivery. For example, '[1] SalesFact (TABLE)'.
ROWS	The identifier and number of data rows delivered using a delivery. For example, '[1] 1250'.

INTERNAL Group

Item	Description
MIN_HASH_TABLE_ SIZE	The minimum size of the hash table, in slots, to prevent resizing of this table during build execution.
PAGE_FAULTS	The number of page faults that occurred while executing the build.
MEMORY	The amount of memory, in megabytes, required to execute the build.

TIMING Group

Item	Description
START	A string value of the form, "Starting Build '<buildname>'".
ELAPSED_TIME	A string value of the form, "<d> days, <h> hours, <m> mins, <s> secs", that gives the total time taken to execute the build.
ELAPSED_SECS	The number of seconds taken to execute the build.
END-SUCCESS	A string value that indicates successful completion of the build. For example, 'Build completed successfully'.
END-FAILURE	A string value that indicates build failure. For example, 'Build failed'.

Examples

As the condition of a condition JobStream node, this example tests whether the last execution of the build called by the JobStream node with ID 3 rejected any data.

```
$WantedNode := NodeAuditID('3');
RETURN (ToInteger(AuditTrail($WantedNode, 'TRANSFORM', 'REJECTED')) = 0)
```

DBMS

Returns the database type for a connection or alias.

These are the values that can be returned:

- CMSRC (IBM Cognos Data Source)
- Published FM Package
- DB2®
- ESSBASE
- INFORMIX
- MSDTS
- ODBC returns the DBMS name retrieved using the ODBC API, for example, Microsoft Access, Microsoft SQL Server, Red Brick® Warehouse
- OLEDB (Microsoft SQL Server using OLE-DB)
- ORACLE
- SAP
- SQLTXT
- SYBASE
- TM1®

Syntax

```
DBMS(<alias>)
```

Symbol	Description
<alias>	A database connection or alias

Examples

```
DBMS('GO_Sales')
```

This example returns SYBASE

DBName

Returns the database name for the alias.

Syntax

```
DBName(<alias>)
```

Symbol	Description
<alias>	A database alias

This example returns GOSales

Examples

```
DBName ('GO_Sales')
```

Delay

Pauses execution for the specified number of seconds.

The default delay is zero seconds. The maximum delay is MAXINT seconds for UNIX systems and MAXINT/1,000 seconds for Windows systems, where MAXINT is the largest value that an integer can store on a particular computer.

This function returns TRUE.

Syntax

```
Delay(<seconds>)
```

Symbol	Description
<seconds>	A positive integer that represents the number of seconds to pause

Examples

```
Delay(60)
```

Driver

Returns the type of driver the database connection or database alias uses.

These are the values that can be returned:

- CMSRC (IBM Cognos Data Source)
- Published FM Package
- DB2
- ESSBASE
- INFORMIX
- MSDTS
- ODBC
- OLEDB (Microsoft SQL Server using OLE-DB)
- ORACLE
- SAP
- SQLTXT
- SYBASE
- TM1

Syntax

```
Driver(<alias>)
```

Symbol	Description
<alias>	A database connection or alias

Examples

```
Driver('GO_Sales')
```

This example returns SYBASE

Exit

Causes IBM Cognos Data Manager to stop execution of the current build, and to show and return the specified reason code.

By default, the returned code is 0.

Entering zero as the code, causes successful completion of the build.

When executing a build within a JobStream, Exit applies only to the individual node. The remainder of the JobStream will proceed as normal.

The returned code has no significance to Cognos Data Manager and is only for use by your scripts or other processes.

Syntax

```
Exit(<code>)
```

Symbol	Description
<code>	An integer

Examples

- `Exit (123)`
- `Exit (0)`

FileCheck

Tests the named file for the specified properties.

This function returns TRUE if all specified properties are TRUE. If not, it returns FALSE.

Syntax

```
FileCheck(<filename> {, <property>})
```

Symbol	Description
<filename>	The full path and file name of a file.
<property>	A string that specifies a property of the file to test. These are the valid properties • 'EXISTS' specifies that IBM Cognos Data Manager can find the file • 'READ' specifies that Cognos Data Manager can read from the file • 'WRITE' specifies that Cognos Data Manager can write to the file • 'EXECUTE' specifies that the file is executable

Examples

```
If (FileCheck('C:\txns\daily.csv', 'EXISTS', 'READ'))
THEN System(DATABUILD -c ODBC 'DSN=MARTLOAD' UpdateMart);
```

If the file, C:\txns\daily.csv exists and Data Manager can read from it, Cognos Data Manager calls DATABUILD to execute the build UpdateMart that resides in the catalog in the ODBC data source MARTLOAD.

FileClose

Closes the file with the specified number. If you specify an invalid file number, FileClose does nothing.

Syntax

```
FileClose(<file_no>)
```

Symbol	Description
<file_no>	The number of the file to close

Note: Use the number returned when you open the file. For more information, see "FileOpen" on page 24.

Examples

```
FileClose($fileno)
```

This example closes the file to which the fileno variable relates.

FileFromParts

Returns a string representation of the specified file, with the specified extension, in the specified location.

Syntax

```
FileFromParts(<directory>,<filename>,<extension>)
```

Symbol	Description
<directory>	The directory for the file
<filename>	The name of the file, without an extension
<extension>	The extension of the file

Notes

- FileFromParts can provide platform independence if you use variables for the parts that differ between operating systems.
- If you set <extension> to NULL, FileFromParts uses .tmp.
- If you set <filename> to NULL, FileFromParts generates a unique file name.
- If you set the location to NULL, FileFromParts uses your default, temporary directory.

Examples

-
```
$file_name := FileFromParts($DS_DATA_DIR, 'results','txt')
```

This example assigns the full file path of results.txt in the IBM Cognos Data Manager data directory. This could resolve to 'C:\Program Files\ibm\cognos\c10\datamanager\data\results.txt' under Windows, or to '/usr/tmp/cognos/data/results.txt' under UNIX.

- `FileFromParts(NULL,NULL,NULL)`

  This example generates the file path of a unique temporary file in your default, temporary directory. You can use the "FileOpen" on page 24 function to create the file.

FileFullPath

Converts a relative file path to an absolute file path.

Syntax

```
FileFullPath(<full_path>)
```

Symbol	Description
<full_path>	A file path, which may be relative

Examples

```
$file_name := FileFullPath('temp.txt')
```

If the current directory is C:\Temp, this example assigns 'C:\Temp\temp.txt' to the file_name variable.

FileList

Returns a comma-separated list of file names. The maximum length of the returned list can be up to 2000 characters in length.

Syntax

```
FileList(<filename> [, <full> [, drill]])
```

Symbol	Description
<filename>	A string value that specifies the files to list. You can include asterisk (*) wildcard characters anywhere in the name of the file, but not in the path.
<full>	This parameter specifies whether to prefix each file with the file path. Set this to TRUE to include file paths and to FALSE to omit them. By default, file paths are omitted.
<drill>	This parameter specifies whether to search subdirectories. Set this to TRUE to search subdirectories and to FALSE to omit them.

Examples

```
FileList('C:\data\*.dat', TRUE)
```

This example returns a comma-separated list of all the files that have the extension .dat and reside in the directory C:\data. Each file is prefixed with the file path.

FileOpen

Opens a file and returns a number to identify the file in subsequent operations.

FileOpen returns NULL if it cannot find the specified file.

Syntax

```
FileOpen(<filename>[, <mode>])
```

Symbol	Description
<filename>	A string value that specifies the files to open.
<mode>	The mode for which you want to open the file • READ specifies open the file for reading. • WRITE specifies open the file for writing, overwriting the file if it exists. • APPEND specifies open an existing file for writing, appending to the file if it exists.

Examples

```
$file_no := FileOpen('C:\Temp\Temp.txt', WRITE);
```

This example opens the file C:\Temp\Temp.txt for writing. Until the file is closed, all subsequent operations on the file should use the number stored in the fileno variable.

FileRead

Reads the next line of text from the specified file.

If the file is empty or the specified file number is invalid, FileRead returns NULL.

If there is no more text to read, FileRead returns NULL.

Syntax

```
FileRead(<file_no>)
```

Symbol	Description
<file_no>	The number of the file from which to read

Examples

```
$TextVar := FileRead($fileno);
```

This example reads a line of text from the file to which the fileno variable points. It assigns this text to the variable TextVar.

FileWrite

Writes one or more lines of text to a file.

If the specified file number is invalid, or the file is open for reading, FileWrite returns NULL. This function may include up to 16 parameters.

Syntax

```
FileWrite(<file_no>[,<value...>])
```

Symbol	Description
<file_no>	The number of the file to which to write.
<value...>	A value of any data type. You can have as many parameters as you require.

Examples

```
$fileno := FileOpen('c:\temp\temp.txt','WRITE');
FileWrite($fileno, 'This is line 1');
FileWrite($fileno, 'This is line 2', 'and line 3');
FileWrite($fileno, 'This is line 4', NULL, 'and line 6');
FileClose($fileno);
$fileno := FileOpen('c:\temp\temp.txt','APPEND');
FileWrite($fileno, This is line 7');
FileClose($fileno);
$fileno := FileOpen('c:\temp\temp.txt', 'READ');
LogMsg(FileRead($fileno));
$textvar := FileRead($fileno);
FileClose($fileno);
```

This code fragment opens the file C:\Temp\temp.txt for writing. The next line writes a line of text to this file. The following line writes two lines (2 and 3) of text. The next line writes three lines of text (4, 5, and 6). The fragment uses NULL to write a line that contains only a carriage return character. The next line of code closes the file.

The next portion of code opens the file for appending, then writes a line of text to the file, then closes it.

The final portion of code opens the file for reading. It writes the first line of the file to the execution log, and reads the second line of the text file to the textvar variable. Finally, the fragment closes the text file.

GetDirectory

Returns the directory portion of a complete file path. If the file path is invalid, GetDirectory returns NULL.

Syntax

```
GetDirectory(<full_path>)
```

Symbol	Description
<full_path>	The path from which to extract directory information

Examples

```
GetDirectory('/usr/tmp/result.txt')
```

This example returns '/usr/tmp/'

GetFileName

Returns the file name portion of a complete file path. If the file path is invalid, GetFileName returns NULL.

Syntax

```
GetFileName(<full_path> [, <strip>])
```

Symbol	Description
<full_path>	The file path from which to extract file name information
<strip>	Specify TRUE or FALSE to indicate whether or not to remove the file extension from the file name

Examples

```
GetFileName('/usr/tmp/result.txt', TRUE)
```

This example returns 'result'

LogMsg

Writes a user message to the build log and returns TRUE.

The message can contain a maximum of 2000 characters.

Syntax

```
LogMsg(<value> [, <value...>])
```

Symbol	Description
<value>	A text string which may contain variables

Examples

```
LogMsg('Phase 1 delivery complete')
```

This example writes "Phase 1 delivery complete" to the build log and returns TRUE.

Lookup

Executes the specified SQL statement on the specified database.

If the statement produces zero rows, Lookup returns NULL.

If the statement produces a single row, Lookup returns that single value.

If the statement produces more than one row, Lookup returns a comma-separated list of values.

If the result table has more than one column, Lookup uses only the left most column.

Syntax

```
Lookup(<alias>, <statement> [, <quote>] [,max_values])
```

Symbol	Description
<alias>	A database connection or alias.
<statement>	An SQL statement.
<quote>	Either TRUE or FALSE to indicate whether or not to enclose each returned value in quotation marks. The default value is FALSE.
<max_values>	Limits the number of items that are retrieved.

Examples

The Product table in the database to which the Sales alias refers contains the following data row.

ProdNo	Description	Price
p0001	Camping Kettle	2.45

```
Lookup('Sales','SELECT Price FROM Product WHERE ProdNo = ''p0001''')
```

This example returns 2.45

MessageCode

Returns the message code of an error or warning message from a fact build, dimension build, or JobStream execution.

An audit ID number identifies the execution process, and a message number identifies the message. For each audit ID, the messages are numbered from 1.

MessageCode returns NULL if the specified message does not exist.

For a full list of error messages, see the IBM Cognos Data Manager message files. These are stored in

- For Windows, by default, \Program Files\ibm\cognos\c10\msgsdk\dmmsgs_en.xml
- For UNIX, *c10_location*/datamanager/message/<message_file>.msg

Each message file contains a group of related numbered messages. For example, the file named auth.msg contains messages related to authorization issues, and the file named exp.msg contains messages related to expressions.

Syntax

```
MessageCode(<audit_id>,<message_no>)
```

Symbol	Description
<audit_id>	The audit ID of the required fact build, dimension build, or JobStream
<message_no>	The number of the message for which the code is required

Examples

```
$msg_code := MessageCode($id,1)
```

This example returns 'DS-DBMS-E402' if the first message specified by the id variable shows that a DBMS driver reported an error.

MessageCount

Returns the number of error and warning messages when a fact build, dimension build, or JobStream is executed.

Syntax

```
MessageCount(<audit_id>)
```

Symbol	Description
<audit_id>	The audit ID of the execution process

Examples

```
$msg_count := MessageCount(19);
```

This example assigns, to the msg_count variable, the number of error or warning messages with audit ID 19.

MessageSeverity

Returns either 'E' or 'W' to indicate whether a particular message is an error or a warning.

Where you give no message number, MessageSeverity returns 'E' if any errors exist, 'W' if any warnings (but no errors) exist, or NULL otherwise.

Syntax

```
MessageSeverity(<audit_id> [, <message_no>])
```

Symbol	Description
<audit_id>	The audit ID of the execution process
<message_no>	The number, starting from 1, of the message

Examples

```
$status := IfNull(MessageSeverity(19),'OK');
```

This example assigns, to the status variable, 'E' if error messages exist with the audit ID 19. It assigns 'W' if warning messages exist. Otherwise, it assigns 'OK' to the status variable.

MessageText

Returns the text of a particular error or warning message when a fact build, dimension build, or JobStream is executed.

If the message does not exist, MessageText returns NULL.

Syntax

```
MessageText(<audit_id>, <message_no>)
```

Symbol	Description
<audit_id>	The audit ID of the execution process to which the message relates
<message_no>	Starting from 1, the number of the message for which the text is required

Examples

```
$msg_txt := MessageText($audit_id,1)
```

This example returns 'DBMS driver ORACLE is not authorized' if the first error or warning message for the execution process specified by the audit_id variable shows that your IBM Cognos Data Manager license does not include the ORACLE DBMS driver.

NodeAuditID

Returns the audit_id of the specified fact build node or dimension build node.

It returns NULL for nodes other than fact build nodes and dimension build nodes, or if there is no fact build node or dimension build node with the specified ID.

Syntax

```
NodeAuditID(<node_id>)
```

Symbol	Description
<node_id>	The identifier of the JobStream fact build or dimension build node.

Examples

```
$WantedNode := NodeAuditID('3');
```

This example assigns to the JobStream variable, the audit_id of build '3' with node ID '3'.

NodeStatus

Returns a character that represents the status of the JobStream node with the specified ID.

The following table gives the possible returned values and their meanings.

Syntax

Value	Meaning
R	The node is processing
S	The node completed successfully
F	The node failed

```
NodeStatus(<node_id>)
```

Symbol	Description
<node_id>	The identifier of a JobStream node

Examples

```
NodeStatus('3')
```

This example returns the status of the JobStream node with the ID '3'.

OpSys

Returns either 'WIN32' or 'UNIX' to indicate the operating system on which IBM Cognos Data Manager is operating.

Syntax

```
OpSys()
```

RowNum

Returns the row number of the member in the fact data collection.

Syntax

```
RowNum()
```

RowsInserted

Returns the number of rows inserted in the specified target table and database by the specified build execution.

By default, this function returns the total of rows inserted in all target tables.

Syntax

```
RowsInserted(<audit_id>[, <table_name>[, <dbalias>]])
```

Symbol	Description
<audit_id>	An integer that identifies a specific execution of a specific fact or dimension build
<table_name>	The name of a target data table
<dbalias>	The name of a database connection

Note: This function returns the number of rows delivered by the build as a whole, not by a particular delivery module.

Examples

```
RowsInserted(9)
```

This example returns the number of rows that the build execution with audit ID 9 inserted into the target tables.

RowsUpdated

Returns the number of rows updated in the specified target table and database by the specified build execution.

By default, this function returns the total number of rows updated in all target tables.

Syntax

```
RowsUpdated(<audit_id>[, <table_name>[, <dbalias>]])
```

Symbol	Description
<audit_id>	An integer that identifies a specific execution of a specific fact or dimension build
<table_name>	The name of a target data table
<dbalias>	The name of a database connection

Examples

```
RowsUpdated(9, 'SalesFact')
```

This example returns the number of rows that the build execution, with audit ID 9, updated in the SalesFact target table.

SendAlert

Writes a user-defined audit record of the type ALERT, into the IBM Cognos Data Manager audit tables, and returns TRUE or FALSE to indicate success or failure.

You can use these records to record specific events that occur during JobStream and build execution. You can also use other tools to access the audit tables, for example, custom audit reports using IBM Cognos Business Intelligence.

Syntax

```
SendAlert(<item>, <value>)
```

Symbol	Description
<item>	The user-defined keyword to which the message relates
<value>	The text of the audit message

Note: Because Cognos Data Manager maintains audit information in the current catalog, you cannot use this function in a file-based project.

Examples

```
SendAlert('START', 'Build 345')
```

SendMail

Sends an email to the specified recipients.

SendMail uses the account of the currently logged on user, and the default email client of the computer on which IBM Cognos Data Manager operates.

If the email is not sent, SendMail returns NULL.

Note: Email attachments are not supported on UNIX.

Syntax

```
SendMail(<profile>,<password>,<subject>,<text>,<send to>[, <copy to>
[, <attachments>]])
```

Symbol	Description
<profile>	The email profile for the computer you are using. You can check this by opening the Control Panel, double-clicking **Mail**, and then clicking **Show Profiles**.
<password>	The password that you use to access email.
<subject>	The subject text of the email.
<text>	The body text of the email. If the text begins with an @ character, a file name is assumed, and the text from the file is used as the basis of the message.
<send to>	A semicolon-delimited list of recipients. SMTP is used as the default protocol. However, you can use any protocol that your email software supports. Other than for SMTP, you must prefix the each address with the protocol specification. For example, FAX:+1(403)2325986.

Symbol	Description
<copy to>	A semicolon-delimited list of recipients of a copy of the email.
<attachments>	A comma-separated list of file paths for up to ten attachments to the email. This is not available under UNIX.

Examples

```
SendMail('MS Exchange Settings', ' ' ,
  'Build Completion',
  'The daily summaries for regional sales have been loaded',
  'ELeblanc@Actup9.com', 'FHillemann@dfd18.com;AM@hgs10.jp',
  'c:\Program Files\ibm\cognos\c10\datamanager\log\summaries.log');
```

This example uses the profile MS Exchange Settings and sends an email to ELeblanc@Actup9.com, copied to FHillemann@dfd18.com and AM@hgs10.jp, with subject "Build Completion" and body text "The daily summaries for regional sales have been loaded". SendMail attaches the file c:\Program Files\ibm\cognos\c10\datamanager\log\summaries.log to the email.

Sql

Executes the specified SQL statement on the specified database connection or alias. Returns TRUE if the statement completes successfully, otherwise causes a script error unless error suppression has been specified.

Syntax

```
Sql(<alias>, <statement> [,<bCognosSQL>],[,<bSuppressErrors>])
```

Symbol	Description
<alias>	A database connection or alias
<statement>	An SQL statement
<bCognosSQL>	The dialect of SQL to use, either native or Cognos SQL. **Note:** You must use Cognos SQL if your SQL statement contains parameters. For more information about Cognos SQL, see the IBM Cognos Data Manager *User Guide*.
<bSuppressErrors>	Allows scripting errors to be suppressed if the SQL fails. If set to TRUE, this function returns FALSE if the SQL fails. Errors validating or connecting to the database alias cannot be suppressed.

Examples

```
SQL('Sales', 'DROP TABLE Temp')
```

This example permanently removes the table named Temp from the Sales database.

System

Executes an operating system command.

IBM Cognos Data Manager passes the command to the operating system unchecked.

Syntax

```
System(<command>)
```

Symbol	Description
<command>	An operating system command

Examples

- On Windows

```
System('del
d:\temp\temp.txt')
System('copy d:\temp\file1.txt d:\temp\file2.txt')
System('dir c:\temp\file2.txt')
```

- On UNIX

```
System('rm
/tmp/temp.txt')
System('cp /tmp/file1.txt /tmp/file2.txt')
```

UUID

Returns a Universally Unique Identifier (UUID) string.

Syntax

```
UUID()
```

VariableInfo

Returns the data type and associated information about a variable.

For any variable, you can use the "TypeInfo" on page 51 function to determine the data type.

For variables of type CHAR, you can determine the maximum string length.

For variables of type NUMBER, you can determine the precision and scale.

Syntax

```
VariableInfo(<variable>, <string>)
```

Symbol	Description
<variable>	The variable for which you want type information
<string>	A string value that specifies the type of information you want

The following table gives the valid settings for <variable>, together with the information that each specifies.

Setting	Description
'DATATYPE'	The function returns a string value that names the data type of the <variable> parameter. Possible values are 'CHAR', 'INTEGER', 'FLOAT', 'NUMBER', 'DATE', 'BOOLEAN', 'TIME', 'BINARY', 'INTERVAL DAY TO SECOND', 'INTERVAL YEAR TO MONTH', 'DATE WITH TIME ZONE', 'TIME WITH TIME ZONE'.
'PRECISION'	For variables of type CHAR, the function returns an integer that gives the maximum number of characters that the variable can store. For variables of type NUMBER, the function returns an integer that gives the maximum number of digits that the variable can contain. For other data types, the function returns zero.
'SCALE'	For variables of type NUMBER, the function returns an integer that gives the maximum number of digits that can follow the decimal point. For other data types, the function returns zero.

Examples

```
VariableInfo(TextVar, 'DATATYPE')
```

If the variable TextVar is of type CHAR, this example returns CHAR

SQL cursor functions

These SQL functions allow you to prepare an SQL statement for execution, open a cursor for the statement, collect data from it, and then close the cursor. The purpose of these functions is to allow multiple processing of rows and columns of data.

Examples

This script illustrates the set of SQL cursor functions used collectively to select data from the monthly_sales table in the Sales database. The script then writes the data to a file named sample.txt.

```
$sqlid := SQLPrepare( 'Sales',
 'select * from monthly_sales where product_name = :p1
order by line_no',
 TRUE );
if $sqlid IS NULL then
begin
    LogMsg( SQLGetLastError() );
    return;
end
$fileid := FileOpen( 'c:\temp\sample.txt', 'write');
```

```
$nColumns := SQLColumnCount( $sqlid );
$linetext := '';
$i := 0;
while $i < $nColumns do
begin
 $i := $i + 1;
 if $i > 1 then $linetext := concat( $linetext, ',' );
 $linetext := concat( $linetext, SQLColumnName( $sqlid,
 $i ) );
end
FileWrite( $fileid, $linetext );
SQLBind( $sqlid, 1, 'sample' );
while SQLFetch( $sqlid ) = 0 do
begin
       $linetext := '';
 $i := 0;
 while $i < $nColumns do
 begin
  $i := $i + 1;
  if $i > 1 then $linetext := concat( $linetext, ',' );
  $linetext := concat( $linetext, '"', tochar( SQLData( $sqlid,
$i ) ),
  '"' );
 end
 FileWrite( $fileid, $linetext );
end
SQLClose( $sqlid );
FileClose( $fileid );
```

SQLPrepare

Prepares an SQL SELECT statement for execution, opens a cursor for the statement, and returns an integer value to identify the cursor for use in subsequent functions.

If there is an error in the SQL statement, SQLPrepare returns NULL.

Syntax

```
SQLPrepare( <alias>, <SQL>, <bCognosSQL> [, <comment>] )
```

Symbol	Description
<alias>	A database connection or alias.
<SQL>	An SQL SELECT statement.
<CognosSQL>	The dialect of SQL to use, either native or Cognos SQL. **Note:** You must use Cognos SQL if your SQL statement contains parameters. For more information about Cognos SQL see the IBM Cognos Data Manager *User Guide*.

Symbol	Description
<comment>	Optional comment about the SELECT statement. This comment is included in the trace information when debugging a build.

Examples

```
$sqlid := SQLPrepare( 'Sales',
 'select * from monthly_sales
 where product_name = :p1 order by line_no', TRUE )
```

This is an extract from the example script in "SQL cursor functions" on page 35. Here, an SQL statement is prepared to select all columns from the monthly_sales table, a cursor is opened, and its value is stored in the sqlid variable. All subsequent operations on the cursor should use the integer value stored in the sqlid variable, until the cursor is closed.

SQLGetLastError

Returns the text of the last error for any SQL statement.

Syntax

```
SQLGetLastError()
```

Examples

```
if $sqlid IS NULL then
begin
    LogMsg( SQLGetLastError() );
    return;
end
```

In this extract, from the example script in "SQL cursor functions" on page 35, if an error is returned for the SQLPrepare statement, the error text is obtained and stored.

SQLColumnCount

After opening a cursor, use SQLColumnCount to find the number of columns returned by the SELECT statement after its execution.

Syntax

```
SQLColumnCount( <CursorId> )
```

Symbol	Description
<CursorId>	The cursor variable assigned to the SELECT statement

Examples

```
SQLColumnCount( $sqlid )
```

In this extract, from the example script in "SQL cursor functions" on page 35, the number of columns found in the monthly_sales table is obtained.

SQLColumnName

Obtains the name of a specific column by column number.

Note: If you know the name of a column, but not the column number, use the "SQLColumnNo" function instead.

Syntax

```
SQLColumnName( <CursorId>, <ColumnNo> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement
<ColumnNo>	The column number for which you want to obtain the column name

Examples

```
$linetext := '';
$i := 0;
while $i < $nColumns do
begin
 $i := $i + 1;
 if $i > 1 then $linetext := concat( $linetext, ',' );
 $linetext := concat( $linetext, SQLColumnName( $sqlid, $i
) );
end
```

In this extract, from the example script in "SQL cursor functions" on page 35, the name of each column is obtained.

SQLColumnNo

Obtains the column number of a specific column by column name.

Note: If you know the number of a column, but not its column name, use the "SQLColumnName" function instead.

Syntax

```
SQLColumnNo( <CursorId>, <ColumnName> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement
<ColumnName>	The column name for which you want to obtain the column number

Examples

```
SQLColumnNo( $cursorid, 'Amount')
```

This example obtains the column number for the Amount column.

SQLBind

Before you can start selecting data for processing, you must bind values to any parameters contained in the SELECT statement.

It is only necessary to prepare a SELECT statement once, but you can bind different values to a bind parameter contained within the statement. You can achieve this by binding the parameter, fetching the data, then rebinding the parameter and fetching a new set of data.

Syntax

```
SQLBind( <CursorId>, <BindNo>, <BindValue> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement
<BindNo>	The parameter number to bind
<BindValue>	The value to bind to a parameter

Examples

```
SQLBind( $sqlid, 1, 'sample' )
```

In this extract, from the example script in "SQL cursor functions" on page 35, the value Sample is bound to parameter 1 within the SELECT statement.

SQLFetch

Fetches a row of data from the executed SELECT statement.

SQLFetch returns

- 0 if it successfully fetches a row of data..
- 100 when all rows have been fetched.
- -1 if there is an error.

Syntax

```
SQLFetch( <CursorId> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement

Examples

```
while SQLFetch( $sqlid ) = 0 do
begin
      $linetext := '';
 $i := 0;
 while $i < $nColumns do
 begin
  $i := $i + 1;
```

```
  if $i > 1
  then $linetext := concat( $linetext, ',' );
  $linetext := concat( $linetext, '"',
  tochar( SQLData( $sqlid, $i ) ), '"' );
 end
 FileWrite( $fileid, $linetext );
end
```

In this example, from the example script in "SQL cursor functions" on page 35, each row of data is fetched.

SQLData

Obtains the value of a specified column in a row of data.

Syntax

```
SQLData( <CursorId>, <ColumnNo> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement
<ColumnNo>	The column number for which to obtain the value

Examples

```
while SQLFetch( $sqlid ) = 0 do
begin
       $linetext := '';
 $i := 0;
 while $i < $nColumns do
 begin
  $i := $i + 1;
  if $i > 1
  then $linetext := concat( $linetext, ',' );
  $linetext := concat( $linetext, '"', tochar( SQLData( $sqlid,
$i ) ), '"' );
 end
 FileWrite( $fileid, $linetext );
end
```

In this extract, from the example script in "SQL cursor functions" on page 35, the value of each column is obtained and used to construct a comma-separated string of values that is written to file.

SQLClose

When you have finished processing data for a SELECT statement, you use SQLClose to close the cursor for the statement.

Syntax

```
SQLClose( <CursorId> )
```

Symbol	Description
<CursorId>	The cursor opened for the SELECT statement

Examples

```
SQLClose( $sqlid );
```

In this extract, from the example script in "SQL cursor functions" on page 35, the cursor for the SELECT statement is closed.

Logical functions

Logical functions return their results dependent upon some test.

Choose

Returns the value that resides at the specified position in a specified list of values. The listed values may be of different data types.

The list may contain up to fifteen values. Choose returns NULL if the specified position is invalid.

Syntax

```
Choose(<index>,<outcome1>[, outcome2...])
<outcome1> ::=
 <outcome2>{, <outcome2>}
```

Symbol	Description
<index>	An integer that denotes which value to return from the list of outcomes
<outcome1>	The value returned if index=1
<outcome2>	The value returned if index=2

Examples

```
Choose(2, 'a','b','c','d')
```

This example returns 'b'

If

Returns the result of one of two expressions, depending on the value of a logical expression.

Syntax

```
If(<value>, <TRUE outcome>, <FALSE outcome>)
```

Symbol	Description
<value>	A logical expression
<TRUE outcome>	The expression returned if <value> evaluates to TRUE
<FALSE outcome>	The expression returned if <value> evaluates to FALSE

Examples

```
If('a'>'b','YES','NO')
```

This example returns 'NO'

IfNull

Returns one of two values. If the first of these is NULL, the second value is returned. Otherwise, the first value is returned.

Syntax

```
IfNull(<value>, <NULL outcome>)
```

Symbol	Description
<value>	A logical expression
<NULL outcome>	The value returned if <value> is NULL

Examples

```
IfNull(myarg,0)
```

This example returns 0 if the value of myarg is NULL or myarg if myarg is not NULL

Mathematical functions

Mathematical functions return the results of mathematical calculations.

Abs

Returns the absolute, or unsigned, value of a number.

Syntax

```
Abs(<number>)
```

Symbol	Description
<number>	Any numeric value

Examples

```
Abs(-1)
```

This example returns 1

Band

Divides a set of numbers into bands of a specified size and returns the ordinal number, commencing with zero, of the band that contains a specified number.

The set of numbers starts with a specified lower bound and has no upper bound.

If the specified number is less than the lower bound, Band returns NULL.

Syntax

```
Band(<number>, <min>, <grain>)
```

Symbol	Description
<number>	A numerical value
<min>	The minimum number into which Band places <number>
<grain>	The size of the bands into which Band divides the set

Examples

- `Band(9, 0, 10)`
  This example returns 0
- `Band(20, 0, 10)`
  This example returns 2
- `Band(20, 10, 10)`
  This example returns 1
- `Band(1, 10, 5)`
  This example returns NULL

Ceil

Returns the smallest integer that is greater than or equal to a specified number.

Syntax

```
Ceil(<number>)
```

Symbol	Description
<number>	A numeric value

Examples

```
Ceil(8.35)
```

This example returns 9

Cos

Returns the cosine of an angle.

Syntax

```
Cos(<number>)
```

Symbol	Description
<number>	A numeric value that represents an angle expressed in radians

Examples

```
Cos(10)
```

This example returns -0.839072

Exp

Returns the mathematical constant raised to the power of the specified number.

Syntax

```
Exp(<number>)
```

Symbol	Description
<number>	A numeric value

Examples

```
Exp(2.5)
```

This example returns 12.182494

Floor

Returns the largest integer that is less than or equal to a specified number.

Syntax

```
Floor(<number>)
```

Symbol	Description
<number>	A numeric value

Examples

```
Floor(8.75)
```

This example returns 8

Ln

Returns the natural logarithm (to base *e*) of the specified number.

Syntax

```
Ln(<number>)
```

Symbol	Description
<number>	A numeric value greater than 0

Examples

```
Ln(10)
```

This example returns 2.302585

Log

Returns the logarithm, to base ten, of a number.

Syntax

```
Log(<number>)
```

Symbol	Description
<number>	A numeric value greater than 0

Examples

```
Log(123)
```

This example returns 2.089905

Mod

Returns the remainder from the integer division of one number by another.

Mod divides the first parameter by the second parameter and returns the remainder.

Syntax

```
Mod(<number>, <number>)
```

Symbol	Description
<number>	An integer
<number>	An integer

Examples

```
Mod(10,3)
```

This example returns 1

Power

Returns the value of one number raised to power of another.

Power raises the first parameter to the power of the second parameter and returns the result.

If the first parameter is negative, the second parameter must be an integer.

Syntax

```
Power(<number>, <number>)
```

Symbol	Description
<number>	A numeric value
<number>	A numeric value

Examples

```
Power(2,3)
```

This example returns 8

Rand

Generates a pseudo-random number.

The generated number is greater than or equal to the specified minimum and less than or equal to the specified maximum.

The generated number is a member of the set {<min> + *n*<grain>} where <min> is the specified minimum number, *n* is some integer, and <grain> is a specified interval.

Optionally, you can provide a seed for the random number generator.

Syntax

```
Rand(<min>, <max>, <grain>[, <seed>])
```

Symbol	Description
<min>	The minimum value of the generated random number
<max>	The maximum value of the generated random number
<grain>	The interval between members of the random number series
<seed>	An optional value with which to seed the random number generator

Examples

```
Rand(2, 10, 2, 5)
```

This example returns a pseudo-random number between 2 and 10 inclusive in steps of 2. Therefore, the result is a member of the set (2,4,6,8,10). IBM Cognos Data Manager seeds its random number generator with the last parameter (5).

Round

Returns the value of a number, rounded to a specified number of decimal places.

By default, IBM Cognos Data Manager rounds to zero decimal places.

If the specified number of decimal places is negative, Cognos Data Manager rounds off digits to the left of the decimal point.

Syntax

```
Round(<number>[, <number>])
```

Symbol	Description
<number>	The number to round
<number>	An integer; the number of decimal places to round

Examples

- `Round(22.653, 1)`
  This example returns 22.7
- `Round(22.653, -1)`
  This example returns 20
- `Round(22.653)`
  This example returns 23

Sign

Indicates whether a number is negative, zero, or positive.

Sign returns -1 if the number is less than zero, Sign returns -1, 0 if the number is zero, and +1 if the number is positives.

Syntax

```
Sign(<number>)
```

Symbol	Description
<number>	A numeric value

Examples

- `Sign(-34)`
  This example returns -1
- `Sign(0)`
  This example returns 0
- `Sign(34)`
  This example returns +1

Sin

Returns the sine of an angle.

Syntax

```
Sin(<number>)
```

Symbol	Description
<number>	An angle expressed in radians

Examples

```
Sin(10)
```

This example returns -0.544021

Sqrt

Returns the square root of a positive number or NULL if the number is negative.

Syntax

```
Sqrt(<number>)
```

Symbol	Description
<number>	A numeric value

Examples

```
Sqrt(12.4)
```

This example returns 3.521363

Tan

Returns the tangent of the specified angle.

Syntax

```
Tan(<number>)
```

Symbol	Description
<number>	An angle expressed in radians

Examples

```
Tan(10)
```

This example returns 0.648361

Trunc

Returns the value of a number, truncated to the specified number of decimal places.

By default Trunc truncates the specified number to zero decimal places.

If the second number parameter is negative, Trunc sets to zero that number of digits to the left of the decimal point.

Syntax

```
Trunc(<number>[, <number>])
```

Symbol	Description
<number>	A numeric value
<number>	An integer value

Examples

- `Trunc(22.653, 1)`
  This example returns 22.6
- `Trunc(22.653, -1)`
  This example returns 20

Member functions

Member functions refer to the IBM Cognos Data Manager dimensional framework, so they are valid only within the context of Cognos Data Manager builds.

You cannot use these functions when acquiring data from an IBM Cognos SQLTXT database.

IsAncestor

Returns a value to indicate whether the specified member is an ancestor of the current member in the specified dimension.

IsAncestor returns TRUE if the specified member is an ancestor of the current member. If the specified member is not an ancestor of the current member, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsAncestor(<dimension>, '<member_id>')
```

Symbol	Description
<dimension>	The name of a dimension element of the current fact build
<member_id>	The identifier of the possible ancestor of the current member

Examples

```
IsAncestor(Period, '2006')
```

This example returns TRUE if the member with identifier '2006' is an ancestor of the current member of the Period dimension. Otherwise, it returns FALSE.

Level

Returns either the name or the business name of the hierarchy level of the current member within the specified dimension.

Syntax

```
Level(<dimension>,'<property>')
```

Symbol	Description
<dimension>	The name of a dimension within the current hierarchy
<property>	The name of the property that this function should return • NAME specifies the name of the level of the current member in the specified dimension • CAPTION specifies the business name of the level of the current member in the specified dimension

Examples

If the current member in the Period hierarchy resides within the level with name 'Qtr' and the business name 'Quarter'

- `Level(Period, 'NAME')`
  This example returns 'Qtr'
- `Level(Period, 'CAPTION')`
  This example returns 'Quarter'

Member

Returns a specified property for the current hierarchy member of the specified dimension.

Syntax

```
Member(<dimension>,'<property>')
```

Symbol	Description
<dimension>	The name of the dimension within the current hierarchy.
<property>	The name of the property to be returned. This can be one of the following properties • ID specifies the identifier of the member. • CAPTION specifies the identifier of the caption of the member. • PARENT specifies the identifier of the parent of the member. • SINGLECHILD specifies TRUE if the member has no siblings; that is, the parent of the member has no other child members. Otherwise, this property evaluates to FALSE.

Examples

If the current member of the Period dimension has the identifier, '200601'

```
Member(Period, 'ID')
```

This example returns '200601'

TypeInfo

Returns data type and associated information about an element of the transformation model.

For any element, you can use the TypeInfo function to determine the data type.

For elements of type CHAR, you can determine the maximum string length.

For elements of type NUMBER, you can determine the precision and scale.

Syntax

```
TypeInfo(<element>, <property>)
<property> ::= 'DATATYPE' | 'PRECISION' | 'SCALE'
```

Symbol	Description
<element>	The transformation model element for which you want type information
<property>	A string value that specifies the type of information you want

The following table gives the valid settings for <property>, together with the information that each setting specifies.

Setting	Description
'DATATYPE'	The function returns a string value that names the data type of the <element> parameter. Possible values are 'CHAR', 'INTEGER', 'FLOAT', 'NUMBER', 'DATE', 'BOOLEAN', 'TIME', 'BINARY', 'INTERVAL DAY TO SECOND', 'INTERVAL YEAR TO MONTH', 'BLOB', 'CLOB', 'DATE WITH TIME ZONE', 'TIME WITH TIME ZONE'.
'PRECISION'	For CHAR elements, the function returns an integer that gives the maximum number of characters that the element can contain. For NUMBER elements, the function returns an integer that gives the maximum number of digits that the element can contain. For other data types, the function returns zero.

Setting	Description
'SCALE'	For NUMBER elements, the function returns an integer that gives the maximum number of digits that can follow the decimal point. For other data types, the function returns zero.

Examples

```
TypeInfo(E1, 'DATATYPE')
```

If element E1 is of type CHAR this example returns 'CHAR'

Unmatched

Returns a Boolean value to indicate whether the current member of the specified dimension is unmatched.

Syntax

```
Unmatched(dimension)
```

Symbol	Description
Dimension	The dimension that includes the member.

Text functions

This topic describes the text functions available.

Char

Returns the ASCII character with the specified code.

Syntax

```
Char(<number>)
```

Symbol	Description
<number>	The numeric code of an ASCII character

Examples

- `Char(65)`

  This example returns 'A'
- `Char(90)`

  This example returns 'Z'

Checksum

Generates a cyclic-redundancy checksum (CRC) from a series of strings.

Use this function to detect changes in the strings. If the CRC value changes, one or more of the parameters have changed.

Note: The CRC can generate, at best, statistically unique (but not totally unique) values across a set of data. It is possible that different sets of data may produce the same checksum value.

Syntax

```
Checksum(<value>[, <value...>])
```

Symbol	Description
<value>	A value of any data type

Examples

```
$CHECK1 := Checksum($RESULT_1, $RESULT_2)
```

Collapse

Returns the string that results from removing all the white space from a specified string.

White space consists of non-printing characters, such as space characters, tab characters, and carriage returns.

Syntax

```
Collapse(<string>)
```

Symbol	Description
<string>	A string value

Examples

```
Collapse('s p a c e s')
```

This example returns 'spaces'

Concat

Returns the string formed from the catenation of two or more values. You can use as many parameters as you require.

Syntax

```
Concat(<value> [, <value...>])
```

Symbol	Description
<value>	A value of any data type

Examples

- `Concat('ware','house')`
  This example returns 'warehouse'
- `Concat(Concat('data',' '),'ware','house')`
  This example returns 'data warehouse'

ConcatSep

Returns the concatenation of two or more strings separated by a specified character or string.

ConcatSep can take up to 16 parameters including the separator.

Syntax

```
ConcatSep(<string>, <value>[, <value...>])
```

Symbol	Description
<string>	A string value; the specified separator
<value>	A value of any data type

Examples

- `ConcatSep('/','a')`
  This example returns 'a'
- `ConcatSep('/','a','b','c')`
  This example returns 'a/b/c'

CountStr

Returns the number of times one string occurs in another.

CountStr searches the first parameter for occurrences of the second parameter and returns the number of times the second parameter occurs in the first.

Syntax

```
CountStr(<string>, <string>)
```

Symbol	Description
<string>	A string value

Examples

```
CountStr('analytically','al')
```

This example returns 2

ExtractStr

Extracts one item from a string that contains a separated list of items.

ExtractStr returns the item that resides in the ordinal position specified.

If the specified ordinal position is greater (or less) than the number of items that are present, ExtractStr returns NULL.

By default, the items are separated with commas, but you can specify an alternative character or string.

Syntax

```
ExtractStr(<string>, <number> [,<string>])
```

Symbol	Description
<string>	A string value that contains a separated list of items
<number>	The ordinal position of the item that you want to extract from <string>
<string>	The character that separates the items within <string>

Examples

If $mystr = 'this,function,extracts,an,item,from,a,list'

- `ExtractStr($mystr, 2)`
  This example returns 'function'
- `ExtractStr($mystr, 5, ',')`
  This example returns 'item'
- `ExtractStr($mystr, 10)`
  This example returns NULL

I18NConvert

Converts a string to the specified encoding.

Syntax

```
I18NConvert(<string>, <encoding>)
```

Symbol	Description
<string>	A string value that contains a separated list of items
<encoding>	The encoding to be used, for example: • UTF-8 (the default) • UTF-16 • US-ASCII • Shift-JIS • ISO-8859-1

Examples

```
tohex( i18nconvert( i18nstring( '€' ), 'utf-16' ) )
```

This example returns the UTF-16 encoding for the euro symbol

I18NString

Defines a string to the specified encoding.

Syntax

```
I18NString(<string>, <encoding>)
```

Symbol	Description
<string>	A string value that contains a separated list of items
<encoding>	The encoding to be used to decode the string, for example: • UTF-8 (the default) • UTF-16 • US-ASCII • Shift-JIS • ISO-8859-1

Examples

- `tohex( i18nstring( '€' ) )`
  This example returns the hex encoding of UTF-8 for the euro symbol
- `tohex( i18nstring( '€', 'utf-16' ) )`
  This example returns the hex encoding of UTF-16 for the euro symbol
- `tohex( i18nstring( '€', 'iso-8859-1' ) )`
  This example returns the hex encoding of iso-8859-1 for the euro symbol
- `i18nstring( '0xE282AC' )`
  This example returns euro symbol which is the UTF-8 encoding for the string

Initcap

Returns the string formed by casting the first character of each word of a specified string to uppercase, and casting the all other characters to lowercase.

Words are delimited by white space or characters that are not alphanumeric.

Syntax

```
Initcap(<string>)
```

Symbol	Description
<string>	A string value

Examples

```
Initcap('the FIRST of december')
```

This example returns 'The First Of December'

InStr

Searches for an occurrence of the second string in the first string. It returns an integer that indicates the position at which the second string occurs in the first.

If InStr finds the second string in the first, it returns the ordinal position, starting at 1, at which the first character of the second string occurs.

If the search is unsuccessful, InStr returns zero.

By default, InStr searches rightward, starting at the left most character of the string to search. However, you can specify to start the search at another position and to search leftward.

To specify a position to start the search, enter the ordinal number of the starting position in the optional third parameter.

To specify a leftward search, enter a negative number in the optional third parameter. If you enter a negative number here, the search commences at the specified number of characters from the right.

By default InStr searches for the first occurrence of the string for which to search. However, you can configure InStr to return the second, third, or other occurrence by specifying the required occurrence in the optional fourth parameter. Note that, if you do this, you must also specify the position at which to start.

Syntax

```
InStr(<string1>,<string2>[, <number1>[, <number2>]])
```

Symbol	Description
<string1>	The string to be searched.
<string2>	The string for which InStr searches.
<number1>	The starting position of the search.
<number2>	An integer that determines for which occurrence of the second string the function searches.

Examples

- `InStr('Lorem ipsum', 'm', -1, 2)`
  This example returns 5
- `InStr('Lorem ipsum', 'm', -1, 3)`
  This example returns 0

IsAlpha

Tests whether a string contains only alphabetic characters.

If the string contains only alphabetic characters; that is, every character is a member of the set ('A' .. 'Z', 'a' .. 'z'), IsAlpha returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsAlpha(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsAlpha('abc123')`
  This example returns FALSE
- `IsAlpha('datamanager')`
  This example returns TRUE
- `IsAlpha('some text')`
  This example returns FALSE

IsAlphaNumeric

Tests whether a string contains only alphabetic and numeric characters.

If the string contains only alphabetic and numeric characters; that is, every character is a member of the set ('A' .. 'Z', 'a' .. 'z', '0' .. '9'), IsAlphaNumeric returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsAlphaNumeric(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsAlphaNumeric('abc123')`
  This example returns TRUE
- `IsAlphaNumeric('abc:123')`
  This example returns FALSE
- `IsAlphaNumeric('abc 123')`
  This example returns FALSE

IsDigit

Tests whether a string contains only numeric characters.

If the string contains only numeric characters, that is, every character is a member of the set ('0' .. '9'), IsDigit returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsDigit(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsDigit('123')`

  This example returns TRUE.
- `IsDigit('-123')`

  This example returns FALSE because '-' is not a numeric character.

IsFloat

Tests whether a string represents a floating-point number.

If the string represents a floating point number, IsFloat returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

If thousand separators are used, they must be applied after every subsequent third digit, and must be logically positioned to allow for the decimal separator.

Syntax

```
IsFloat(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsFloat('12.0')`

  This example returns TRUE
- `IsFloat('12')`

  This example returns FALSE

IsInteger

Tests whether a string represents an integer.

If the string represents an integer, IsInteger returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

If thousand separators are used, they must be applied after every subsequent third digit, and must be logically positioned to allow for the decimal separator.

Syntax

```
IsInteger(<string>)
```

Symbol	Description
<string>	A string value

Example

- `IsInteger('12.0')`
  This example returns FALSE
- `IsInteger('12')`
  This example returns TRUE

IsLower

Tests whether a string consists of only lowercase, alphabetic characters.

If the string contains only lowercase, alphabetic characters, that is, every character is a member of the set ('a' .. 'z'), IsLower returns TRUE. Otherwise, it returns FALSE. If the string contains one or more spaces, hyphens, apostrophes, commas, or other punctuation, IsLower returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsLower(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsLower('abc')`
  This example returns TRUE
- `IsLower('a b c')`
  This example returns FALSE

IsNumeric

Tests whether a string represents a numeric value.

If the string represents a numeric value, IsNumeric returns TRUE. Otherwise, it returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

If thousand separators are used, they must be applied after every subsequent third digit.

Syntax

```
IsNumeric(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsNumeric('123')`
  This example returns TRUE
- `IsNumeric('-123')`
  This example returns TRUE

IsUpper

Tests whether a string consist of only uppercase, alphabetic characters.

If the string contains only uppercase, alphabetic characters; that is, every character is a member of the set ('A' .. 'Z'), IsUpper returns TRUE. Otherwise, it returns FALSE. If the string contains one or more spaces, hyphens, apostrophes, commas, or other punctuation, IsUpper returns FALSE.

Depending on the target DBMS, IBM Cognos Data Manager may represent TRUE with a non-zero, numeric value, and FALSE with zero.

Syntax

```
IsUpper(<string>)
```

Symbol	Description
<string>	A string value

Examples

- `IsUpper('ABC')`
  This example returns TRUE
- `IsUpper('A, B, C')`
  This example returns FALSE

Left

Returns the left most specified number of characters of a string.

If the string contains fewer than the specified number of characters, Left returns the original string.

Syntax

```
Left(<value>, <number>)
```

Symbol	Description
<value>	A string value
<number>	The number of characters to return

Examples

- `Left('catalog', 3)`
  This example returns 'cat'
- `Left('text', 7)`
  This example returns 'text'

Length

Returns the length (number of characters) of the string representation of a value.

Syntax

```
Length(<value>)
```

Symbol	Description
<value>	A value of any data type

Examples

```
Length('text')
```

This example returns 4

Lower

Casts a string to lowercase.

Syntax

```
Lower(<string>)
```

Symbol	Description
<string>	A string value

Examples

```
Lower('This Is Some Text')
```

This example returns 'this is some text'

LPad

LPad adds (or pads) a specified string to form a string of the specified length.

By default, LPad adds or pads the string with space characters, but you can specify a different character or character sequence.

If the specified string contains more characters than the specified length, LPad returns the left most specified number of characters of the string.

Syntax

```
LPad(<string>, <number>[, <string>])
```

Symbol	Description
<string>	A string value
<number>	The length (in number of characters) of the returned string
<string>	The character (or character sequence) with which to left-pad the first string

Examples

- `LPad('x',5,'*')`
  This example returns '****x'
- `Lpad('text',11,'the')`
  This example returns 'thethettext'
- `Lpad('textual', 4)`
  This example returns 'text'

LTrim

Removes specified characters from the left of a specified string.

This function removes characters from the left of the specified string up to, but not including, the left most character that is not a member of the specified set of characters.

By default, the set of characters to remove includes only the space character, which causes LTrim to remove all leading spaces. However, you can specify another set.

Syntax

```
LTrim(<string>[, <string>])
```

Symbol	Description
<string>	A string value
<string>	A string value that gives the set of characters to remove from the first string

Examples

- `LTrim('   text')`
  This example returns 'text'
- `LTrim('pdpcPpdEND','pdc')`

This example returns 'PpdEND'

Replace

Returns the string formed by replacing all instances of one character sequence with another in the specified string.

If you omit the replacement character sequence, Replace removes all instances of the specified sequence from the specified string.

Syntax

```
Replace(<string>, <search>[, <replace>])
```

Symbol	Description
<string>	A string value.
<search>	The string value for which to search.
<replace>	The string value that replaces <search>. By default this is an empty string.

Examples

- `Replace('BUS HOP','H','ST')`

  This example returns 'BUS STOP'
- `Replace('BUS HOP', 'H')`

  This example returns 'BUS OP'

Right

Returns the right most specified number of characters of a string.

If the string contains fewer than the specified number of characters, Right returns the entire string.

Syntax

```
Right(<string>, <number)
```

Symbol	Description
<string>	A string value
<number>	The number of characters to return

Examples

- `Right('catalog', 3)`

  This example returns 'log'
- `Right('text', 7)`

  This example returns 'text'

RPad

RPad adds (or pads) a specified string to form a string of the specified length.

By default, RPad adds or pads the string with space characters, but you can specify a different character or character sequence.

If the specified string contains more characters than the specified length, RPad returns the left most specified number of characters of the string.

Syntax

```
RPad(<string>, <number>[, <string>])
```

Symbol	Description
<string>	A string value
<number>	The length (in number of characters) of the returned string
<string>	The character (or character sequence) with which to right-pad the first string

Examples

- `RPad('x',5,'*')`
  This example returns 'x****'
- `RPad('text',11,'the')`
  This example returns 'textthethet'
- `RPad('textual',4,'*')`
  This example returns 'text'

RTrim

Removes specified characters from the right of a specified string.

This function removes characters from the right of the specified string up to, but not including, the right most character that is not a member of the specified set of characters.

By default, the set of characters to remove includes only the space character, which causes RTrim to remove all trailing spaces. However, you can specify another set.

Syntax

```
RTrim(<string>[,<string>])
```

Symbol	Description
<string>	A string value
<string>	A string value that gives the set of characters to remove from the first string

Examples

```
RTrim('BATESpeGepp','pe')
```

This example returns 'BATESpeG'

Soundex

Returns the Oracle-compatible sound specification of a string.

Syntax

```
Soundex(<string>)
```

Symbol	Description
<string>	A string value

Examples

```
Soundex('word')
```

This example returns W630

SubStr

Returns the specified portion (or substring) of a string.

The returned string is the portion that starts at the specified ordinal position and continues for the specified number of characters.

If the starting position is positive, SubStr counts from the left, with the left most character being at position 1. However, if the starting position is negative, SubStr counts from the right, with the right most character being at position 1. In either case, SubStr returns the substring to the right of the starting position.

If you do not specify a number of characters, SubStr returns the portion of the original string to the right of, and including, the starting position.

If there are fewer than the specified number of characters, SubStr returns the portion to the right of, and including, the starting position.

Syntax

```
SubStr(<string>, <from>[, <count>])
```

Symbol	Description
<string>	A string value
<from>	An integer that gives the position within <string> at which SubStr begins the returned string
<count>	An integer that gives the required length of the returned string

Examples

- `SubStr('textual',1,4)`
  This example returns 'text'
- `SubStr('textual',-7,4)`
  This example returns 'text'
- `SubStr('textual',6,5)`
  This example returns 'al'

Translate

Processes a string by replacing all characters that appear in one set with the corresponding characters from another.

Syntax

```
Translate(<string>, <old>[, <new>])
```

Symbol	Description
<string>	A string value
<old>	A string value that contains the characters to be translated
<new>	A string value that provides the characters to replace the matching characters on <old>

Notes

- The length of <old> must be greater than or equal to the length of <new>
- Translate removes from the returned string any character that appears in <old> that has no corresponding character in <new>

Examples

- ```
  Translate('E746VBW',
            '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
            '9999999999XXXXXXXXXXXXXXXXXXXXXXXXXX')
  ```
  This example returns 'X999XXX'
- `Translate('E746VBW','0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ','0123456789')`
  This example returns '746'

Trim

Returns the string formed by removing specified characters from the left and right of a string.

Syntax

```
Trim(<string>[, <string>])
```

Symbol	Description
<string>	A string value
```

Symbol	Description
<string>	A string value that gives the set of characters to remove from the first string

Notes

- Trim removes characters from the left and right of <string> up to, but not including, the first character that is not a member of the specified set of characters.
- By default, the set of characters to remove consists of the space characters, so that all leading and trailing spaces are removed.
- This function is equivalent to LTrim(RTrim(<string>, <charset>), <charset>).

Examples

- `Trim('   text   ')`

  This example returns 'text'
- `Trim('pdpcPpdENDpdp','pdc')`

  This example returns 'PpdEND'

Upper

Casts a string to uppercase.

Syntax

```
Upper(<string>)
```

Symbol	Description
<string>	A string value

Examples

```
Upper('this is some text.')
```

This example returns 'THIS IS SOME TEXT'

Date functions

Date functions perform calculations on date values, return date values, or both.

You cannot pass dates as text representations. To pass a text representation of a date to a date function, use the “ToDate” on page 4 function to convert the text to a date value.

You must specify a valid date when using a date function. If you specify an invalid date, IBM Cognos Data Manager returns an error. For example, 2006-04-31 is an invalid date.

AddDaysToInterval

Returns a day to second interval resulting from adding the specified number of days to a day to second interval.

Syntax

```
AddDaysToInterval(<interval>, <number>)
```

Symbol	Description
<interval>	A day to second interval value
<number>	An integer that gives the number of days to add to <interval>

Examples

- `AddDaysToInterval(ToIntervalDS('200 07:00:00.12'),5)`
  This example returns '205 07:00:00.12'
- `AddDaysToInterval(ToIntervalDS('200 07:00:00.99'),-5)`
  This example returns '195 07:00:00.99'
- `AddDaysToInterval(ToIntervalDS('200 07:00:00'),75)`
  This example returns '275 07:00:00'
- `AddDaysToInterval(ToIntervalDS('200 00:00:00'),-250)`
  This example returns '-50 00:00:00'
- `AddDaysToInterval(ToIntervalDS('200-070000.123'), sddd-hh:miss.fff),5)`
  This example returns '205 07:00:00.123'
- `AddDaysToInterval(ToIntervalDS('-200 00:00:00'),-50)`
  This example returns '-250 00:00:00'

AddMonthsToDate

Returns a date or date with time zone resulting from adding the specified number of months to a date or date with time zone.

Syntax

```
AddMonthsToDate(<date>, <number>)
```

Symbol	Description
<date>	A date or date with time zone value
<number>	An integer that gives the number of months to add to <date>

Examples

- `AddMonthsToDate(ToDate('2006-07-09','yyyy-mm-dd'),2)`
  This example returns 00:00:00 on 09 September 2006
- `AddMonthsToDate(ToDate('2006-01-30','yyyy-mm-dd'),1)`
  This example returns 00:00:00 on 28 February 2006

AddMonthsToInterval

Returns a year to month interval resulting from adding the specified number of months to a year to month interval.

Syntax

```
AddMonthsToInterval(<interval>, <number>)
```

Symbol	Description
<interval>	A year to month interval value
<number>	An integer that gives the number of months to add to <interval>

Examples

- `AddMonthsToInterval(ToIntervalYM('200-07'),5)`
  This example returns 201-00
- `AddMonthsToInterval(ToIntervalYM('200-07'),-8)`
  This example returns 199-11
- `AddMonthsToInterval(ToIntervalYM('-200-07'),-8)`
  This example returns -201-03
- `AddMonthsToInterval(ToIntervalYM('200 07','syyyy mm'),2)`
  This example returns 200-09
- `AddMonthsToInterval(ToIntervalYM('-200 07','syyyy mm'),2)`
  This example returns 200-05

AddSecondsToInterval

Returns a day to second interval resulting from adding the specified number of seconds to a day to second interval.

Syntax

```
AddSecondsToInterval(<interval>, <number>)
```

Symbol	Description
<interval>	A day to second interval value
<number>	A numeric value that gives the number of seconds to add to <interval>

Examples

- `AddSecondsToInterval(ToIntervalDS('200 07:00:00.12'),5)`
  This example returns '200 07:00:05.12'
- `AddSecondsToInterval(ToIntervalDS('200 07:00:00.99'),-5)`
  This example returns '200 06:59:55.99'
- `AddSecondsToInterval(ToIntervalDS('200 07:00:00'),75.123)`
  This example returns '200 07:01:15.123'
- `AddSecondsToInterval(ToIntervalDS('200 00:00:00'),-50)`
  This example returns '199 23:59:10'
- `AddSecondsToInterval(ToIntervalDS('200-070000.123'), sddd-hhmiss.fff),5.12)`
  This example returns '200 07:00:05.243'

- `AddSecondsToInterval(ToIntervalDS('-200 00:00:00'),-50)`
  This example returns '-200 00:00:50'

AddToDate

Returns a date or date with time zone resulting from adding the specified number of days to a date or date with time zone.

Syntax

```
AddToDate(<date>, <number|interval>)
```

Symbol	Description
<date>	A date or date with time zone value
<number|interval>	An integer that gives the number of days to add to <date> or an interval value

Examples

- `AddToDate(ToDate('2006-07-09','yyyy-mm-dd'),5)`
  This example returns 00:00:00 on 14 July 2006
- `AddToDate('2007-12-12 00:00:01.4', ToIntervalDS( '-0 00:00:01.8' ))`
  This example subtracts 1.8 seconds from a date and returns 2007-12-11 23:59:59.6

AddYearsToDate

Returns a date or date with time zone resulting from adding the specified number of years to a date or date with time zone.

Syntax

```
AddYearsToDate(<date>, <number>)
```

Symbol	Description
<date>	A date or date with time zone value
<number>	An integer that gives the number of years to add to <date>

Examples

- `AddYearsToDate(ToDate('2002-07-09','yyyy-mm-dd'),3)`
  This example returns 00:00:00 on 09 July 2005
- `AddYearsToDate(ToDate('2005-02-29','yyyy-mm-dd'),1)`
  This example returns 00:00:00 on 28 February 2006

AddYearsToInterval

Returns a year to month interval resulting from adding the specified number of years to a year to month interval.

Syntax

```
AddYearsToInterval(<interval>, <number>)
```

Symbol	Description
<interval>	A year to month interval value
<number>	An integer that gives the number of years to add to <interval>

Examples

- `AddYearsToInterval(ToIntervalYM('200-07'),5)`
  This example returns 205-07
- `AddYearsToInterval(ToIntervalYM('200-07'),-8)`
  This example returns 192-07
- `AddYearsToInterval(ToIntervalYM('-200-07'),-8)`
  This example returns -208-07
- `AddYearsToInterval(ToIntervalYM('200 07','syyyy mm'),2)`
  This example returns 202-07
- `AddYearsToInterval(ToIntervalYM('-200 07','syyyy mm'),2)`
  This example returns -198-07
- `AddYearsToInterval(ToIntervalYM('2-01'),-3)`
  This example returns -0-11

DaysBetween

Returns the number of days between two specified dates, dates with time zone, or day to second interval values.

Syntax

```
DaysBetween(<date|interval1>, <date|interval2>)
```

Symbol	Description
<date | interval1>	A date, date with time zone, or day to second interval value
<date | interval2>	A date or day to second interval value (this must be the same data type as <date | interval1>)

Notes

- If <date | interval2> is greater than <date | interval1>, DaysBetween returns a negative value. Otherwise, it returns a positive value.
- DaysBetween ignores the hours, minutes, and seconds parts of values.

Examples

If dv1 is 1 January 2006 and dv2 is 20 January 2006

- `DaysBetween(dv2,dv1)`
  This example returns 19
- `DaysBetween(dv1,dv2)`

This example returns -19

- `DaysBetween(ToInterval('20 00:00:10'), ToInterval('19 23:59:10'))`
  This example returns 1
- `DaysBetween(ToInterval('20 00:00:10'), ToInterval('21 00:01:50'))`
  This example returns -1
- `DaysBetween(ToInterval('-20 00:00:10'), ToInterval('-19 23:59:10'))`
  This example returns -1
- `DaysBetween(ToInterval('-20 00:00:10'), ToInterval('-21 00:01:50'))`
  This example returns 1

FirstOfMonth

Returns the first day of the month.

Syntax

```
FirstOfMonth(<date>)
```

Symbol	Description
<date>	A date or date with time zone value

Examples

```
FirstOfMonth(ToDate('2005-07-09')
```

This example returns 2005-07-01

IsLeapYear

Returns TRUE or FALSE to indicate whether a specified date is in a leap year.

Syntax

```
IsLeapYear(<date>)
```

Symbol	Description
<date>	A date or date with time zone value

Note: This function returns TRUE if the specified date is in a leap year. Otherwise, it returns FALSE.

Examples

- `IsLeapYear(ToDate('2004-07-12','yyyy-mm-dd'))`
  This example returns TRUE
- `IsLeapYear(ToDate('2005-07-12','yyyy-mm-dd'))`
  This example returns FALSE

IsLeapYearDay

Returns TRUE or FALSE to indicate whether a specified date is a leap year day.

Syntax

```
IsLeapYearDay(<date>)
```

Symbol	Description
<date>	A date or date with time zone value

Note: This function returns TRUE if <date> is February 29 of a leap year. Otherwise, it returns FALSE.

Examples

- `IsLeapYearDay(ToDate('2004-02-29','yyyy-mm-dd'))`
  This example returns TRUE
- `IsLeapYearDay(ToDate('2004-02-28','yyyy-mm-dd'))`
  This example returns FALSE

IsValidDate

Returns TRUE or FALSE to indicate whether a string represents a valid date in the specified format.

Syntax

```
IsValidDate(<string>[, <format>])
```

Symbol	Description
<string>	A string value.
<format>	A string value that gives the date format against which to test <string>. By default, IBM Cognos Data Manager assumes the date format to be 'yyyy-mm-dd' or 'yy-mm-dd', depending on the number of digits presented.

Examples

- `IsValidDate('2005-10-02','yyyy-mm-dd')`
  This example returns TRUE
- `IsValidDate('2005-10-32','yyyy-mm-dd')`
  This example returns FALSE
- `IsValidDate('2005-10-20')`
  This example returns TRUE

IsValidIntervalDS

Returns TRUE or FALSE to indicate whether a value represents a valid day to second interval.

The input value can be of type CHAR, INTERVAL DAY TO SECONDS, or any numeric data type representing the number of seconds.

For input values of type CHAR, you can specify the interval format. If you omit a format, the input value must be in the default IBM Cognos Data Manager day to

second interval format of sddddddddd hh:mi:ss[,fffffffff]. The fractions of a second part is optional and can be less than the maximum precision of 9. The number of days can be less than the maximum precision of 9.

For numeric values, IsValidInterval returns TRUE for values in the range -86399999999999.999999999 to 86399999999999.999999999, otherwise returns FALSE.

IsValidIntervalDS returns TRUE for values of type INTERVAL DAY TO SECOND.

Syntax

```
IsValidIntervalDS(<string>[, <format>]))
```

Symbol	Description
<string>	A text representation of the interval
<format>	The format of <string>

Examples

- `IsValidIntervalDS ('1 121314123','sddd hhmissfff')`
  This example returns TRUE
- `IsValidIntervalDS ('-100 12:13:14.123')`
  This example returns TRUE
- `IsValidIntervalDS (100.999)`
  This example returns TRUE
- `IsValidIntervalDS ('1 121314123')`
  This example returns FALSE
- `IsValidIntervalDS ('-100 12:13:14.123', 'sddd hhmissfff')`
  This example returns FALSE
- `IsValidIntervalDS ('1000 121314123', 'sddd hhmissfff')`
  This example returns FALSE
- `IsValidIntervalDS (86400000000000)`
  This example returns FALSE

IsValidIntervalYM

Returns TRUE or FALSE to indicate whether a value represents a valid year to month interval.

The input value can be of type CHAR, INTERVAL YEAR TO MONTH, or INTEGER.

For input values of type CHAR, you can specify the interval format. If you omit a format, the input value must be in the default IBM Cognos Data Manager year to month interval format of syyyyyyyyy-mm. the year part can be less than the maximum precision of 9.

For input values of type INTEGER, IsValidIntervalYM returns TRUE for values in the range -11999999999 to 11999999999.

IsValidIntervalYM returns TRUE for values of type INTERVAL YEAR TO MONTH.

Syntax

```
IsValidIntervalYM(<string>[, <format>]))
```

Symbol	Description
<string>	A text representation of the interval
<format>	The format of <string>

Examples

- `IsValidIntervalYM ('1 11', 'syyy mm')`
  This example returns TRUE
- `IsValidIntervalYM ('1-11')`
  This example returns TRUE
- `IsValidIntervalYM (40)`
  This example returns TRUE
- `IsValidIntervalYM (-40)`
  This example returns TRUE
- `IsValidIntervalYM ('1 11')`
  This example returns FALSE
- `IsValidIntervalYM ('1-11', 'syyy mm')`
  This example returns FALSE
- `IsValidIntervalYM ('+9999 11', 'syyy mm')`
  This example returns FALSE

IsValidTime

Returns TRUE or FALSE to indicate whether a value represents a valid time.

The input value can be of type CHAR, TIME, TIME WITH TIME ZONE or any numeric data type.

For input values of type CHAR, you can specify the time format. If you omit a format, the input value must be in the default IBM Cognos Data Manager time format hh:mi:ss[,fffffffff]. The fractions of a second part is optional and can be less than the maximum precision of 9.

Numeric values must be in the range 0-86399.999999999

IsValidTime returns TRUE for values of type TIME.

Syntax

```
IsValidTime(<value>[, <format>]))
```

Symbol	Description
<value>	A text representation of the time or TIME, TIME WITH TIME ZONE, or any numeric data type
<format>	The format of <value> if <value> is a string

Examples

- `IsValidTime ('121314.1234','hhmiss.ffff')`
  This example returns TRUE
- `IsValidTime ('121314.1234-5:00','hhmiss.ffffstzh:tzm')`
  This example returns TRUE
- `IsValidTime ('12:13:14')`
  This example returns TRUE
- `IsValidTime (86399.99)`
  This example returns TRUE
- `IsValidTime ('121314')`
  This example returns FALSE
- `IsValidTime (-4)`
  This example returns FALSE
- `IsValidTime (90000)`
  This example returns FALSE

LastOfMonth

Returns the last day of the month.

Syntax

```
LastOfMonth(<date>)
```

Symbol	Description
<date>	A date or date with time zone value

Examples

```
LastOfMonth(ToDate('2005-07-09'))
```

This example returns 2005-07-31

MonthsBetween

Returns the number of months between two date, date with time zone, or year to month interval values.

Syntax

```
MonthsBetween(<date|interval1>, <date|interval2>)
```

Symbol	Description
<date|interval1>	A date, date with time zone, or year to month interval value
<date|interval2>	A date, date with time zone, or year to month interval value (this must be the same data type as <date|interval1>

Notes

- If <date|interval2> is greater than <date|interval1>, MonthsBetween returns a negative value. Otherwise, it returns a positive value.

- MonthsBetween converts both values to months and returns <date | interval1> - <date | interval2>

Examples

- `MonthsBetween(ToIntervalYM('20-10'), ToIntervalYM('19-10'))`

  This example returns 12
- `MonthsBetween(ToIntervalYM('20-10'), ToIntervalYM('21-10'))`

  This example returns -12
- `MonthsBetween(ToIntervalYM('-20-10'), ToIntervalYM('-19-10'))`

  This example returns -12
- `MonthsBetween(ToIntervalYM('-20-10'), ToIntervalYM('-21-10'))`

  This example returns 12

SecondsBetween

Returns the number of seconds between two date, time, date with time zone, time with time zone, or day to second interval values.

Syntax

`SecondsBetween(<date|time|interval1>, <date|time|interval2>)`

Symbol	Description
<date | time | interval1>	A date, time, date with time zone, time with time zone, or day to second interval value
<date | time | interval2>	A date, time, date with time zone, time with time zone, or day to second interval value (this must be the same data type as <date | time | interval1>

Notes

- SecondsBetween converts both values to seconds and returns <date | time | interval1> - <date | time | interval2>.
- If <date | time | interval1> or <date | time | interval2> are dates with no time, then the time of each value is presumed to be zero seconds after midnight. For example, 2005-07-01 is equivalent to 2005-07-01 00:00:00.

Examples

In these examples, myDate is ten seconds after midday on July 1 2005 and that the system time is ten minutes after midday on July 1 2005.

- `SecondsBetween(SysDate(), myDate)`

  This example returns 590
- `SecondsBetween(myDate, SysDate())`

  This example returns -590
- `$dval := ToDate('2005-07-01 12:00:00', 'yyyy-mm-dd hh:mi:ss'); RETURN SecondsBetween(myDate, $dval)`

  This example fragment returns 10
- `SecondsBetween(ToTime('10:00:10.123'), ToTime('09:59:50'))`

  This example returns 20.123

- `SecondsBetween(ToTime('10:00:10'), ToTime('10:01:50'))`
  This example returns -100
- `SecondsBetween(ToInterval('20 00:00:10'), ToInterval('19 23:59:10'))`
  This example returns 20
- `SecondsBetween(ToInterval('20 00:00:10'), ToInterval('20 00:01:50'))`
  This example returns -100
- `SecondsBetween(ToInterval('-20 00:00:10'), ToInterval(-19 23:59:10'))`
  This example returns 20
- `SecondsBetween(ToInterval('-20 00:00:10'), ToInterval('-20 00:01:50'))`
  This example returns -100

SysDate

Returns the current system date and time using the optional format.

If no format is provided, 'yyyy-mm-dd hh:mi:ss' is used, and the data type of the returned value is DATE, otherwise it is CHAR.

Syntax

```
SysDate([<format>])
```

Symbol	Description
<format>	A string value that specifies a date format

Examples

```
SysDate('dd-mmm-yyyy hh:mi:ss')
```

If the current system time is twenty minutes and forty-five seconds past ten on the morning of June 4 2005, this example returns '04-Jun-2005 10:20:45'

Chapter 2. The IBM Cognos Data Manager scripting language

The IBM Cognos Data Manager scripting language is a procedural language that you can use in the definition of specific items.

The items for which you can use the scripting language are as follows:

- derivations
- internal user-defined functions
- output filters
- JobStream procedure nodes and condition nodes
- conditions for SQL statement WHERE clauses in IBM Cognos SQLTXT only
- Cognos SQLTXT column expressions

It supports Cognos Data Manager data processing and data mart maintenance. It is not a general programming language.

A script consists of a series of statements and statement blocks, with a semicolon terminating each statement. Statement blocks commence with the BEGIN keyword and end with the END keyword, like this:

```
BEGIN
  $Counter := $Counter + 1;
  $Result := $Result * $Counter;
END
```

Carriage return, tab, and space characters are not significant. As with most free-form syntax, you can use new lines and indentation to aid readability.

You can insert comments at any point in a script. Comments start with '//' and continue to the end of the current line.

Assignment operator in scripts

Use the assignment operator (:=) to assign a value to a variable.

The value can be a literal expression, the value of another variable, or the result of an expression. For example:

```
$TextVar_1 := 'Assignment Example';
$TextVar_2 := $TextVar_1;
$NumVar_3 := ($NumVar_2 - $NumVar_1)/$NumVar_1;
```

Note: When referring to a variable, you must prefix the name of the variable with a dollar symbol ($). This differentiates variables from function arguments and transformation model elements.

Returned value in scripts

Use a RETURN statement to return a value and terminate the script.

Each script can return one value on termination. For example:

```
RETURN $Result;
```

Because the RETURN keyword terminates the script, any statements that follow a RETURN statement are ignored.

Comparison of values in scripts

Generally, IBM Cognos Data Manager uses standard conventions to compare values.

Numeric values

IBM Cognos Data Manager compares numeric values arithmetically.

For example, 10 > 2.

Dates and times

IBM Cognos Data Manager considers earlier dates and times less than later dates and times.

For example, Jun-22-2005 10:20:24 < Jun-30-2005 09:15:47.

Characters and strings

IBM Cognos Data Manager compares characters by comparing their character codes (either ASCII or EBCDIC).

For example, 'A' has ASCII code 65; 'a' has ASCII code 97. Therefore, 'a' > 'A'.

When comparing string values, Cognos Data Manager compares the left most character in each string. If these characters are equal, Cognos Data Manager compares the next character in each string. This continues until Cognos Data Manager finds a difference, or until there are no more characters to check, in one or both strings. If Cognos Data Manager finds a difference, it returns the result of the last character comparison. For example, 'Text'> 'TEXT'.

If the strings are of unequal length, and each character in the shorter string matches the corresponding character in the longer string, the longer string is the greater.

For example:

- '10' < '2' (because '1' < '2')
- 'Text' > 'TEXT' (because 'e' > 'E')
- 'Text' < 'Textual' (because 'Textual' is longer)

NULL values

NULL is a special case. Whenever you make a comparison with NULL, all operators (except IS) return NULL.

For example, the comparison NULL = NULL returns NULL. Therefore, you may want to include a test for NULL if the operand may contain this value.

For example:

```
IF(myvar IS NULL, 0, IF(myvar < 0, 0, myvar))
```

If myvar is NULL, this expression returns 0. Otherwise, it returns the result of the inner IF function.

Without the test for null, the expression would be as follows:

```
IF(myvar < 0, 0, myvar)
```

If myvar were NULL, myvar < 0 would fail. Therefore, the expression would return NULL which is the value of myvar.

Operators

There are two types of operators in IBM Cognos Data Manager: logical and mathematical.

Logical operators

Binary logical operators compare two values. Unary logical operators operate on a single value. The result of a logical operation is either TRUE or FALSE.

In IBM Cognos Data Manager, TRUE is the Boolean value known as True, Yes, and Set; FALSE is the Boolean value known as False, No, and Clear.

Operator	Description
=	Equals. This binary operator returns TRUE if the operands to its left and right are equal. Otherwise, it returns FALSE. **Examples** • 1=2 returns FALSE • 3=3 returns TRUE
!=	Not equals. This binary operator returns TRUE if the operands to its left and right are not equal. Otherwise it returns FALSE. **Examples** • 1!=2 returns TRUE • 3!=3 returns FALSE
<>	This binary operator is equivalent to !=.
>=	Greater than or equals. This binary operator returns TRUE if the operand to its left has a value greater than or equal to that of the operand to its right. Otherwise, it returns FALSE. **Examples** • 2>=1 returns TRUE • 3>=3 returns TRUE • 2>=3 returns FALSE
>	Greater than. This binary operator returns TRUE if the operand to its left has a value greater than that of the operand to its right. Otherwise, its returns FALSE. **Examples** • 2>1 returns TRUE • 3>3 returns FALSE

Operator	Description
<=	Less than or equals. This binary operator returns TRUE if the operand to its left has a value less than or equal to that of the operand to its right. Otherwise, it returns FALSE. **Examples** • 1<=2 returns TRUE • 2<=2 returns TRUE • 3<=2 returns FALSE
<	Less than. This binary operator returns TRUE if the operand to its left has a value less than that of the operand to its right. Otherwise, it returns FALSE. **Examples** • 1<2 returns TRUE • 2<2 returns FALSE
IS	Is. This operator tests for NULL values. The right operand must always be either NULL or NOT NULL. When used with NULL, this operator returns TRUE if the left operand is NULL; otherwise it returns FALSE. When used with NOT NULL, this operator returns FALSE if the left operand is NULL. Otherwise it returns TRUE. **Examples** If myvar is NULL • myvar IS NULL returns TRUE • 'abc' is NOT NULL returns TRUE • 'xyz' IS NULL returns FALSE
IN	In. This binary operator takes a single value as its left operand and a list as its right operand. The list can be either a literal list or derived from an SQL SELECT statement. This operator returns TRUE if the value of the left operand exists within the list. Otherwise, it returns FALSE. **Examples** 2 IN (1,2,3,4) returns TRUE
BETWEEN	Between. This binary operator takes a single value as its left operand, and a range as its right operand. The range must be of the form x AND y, where x<y. This operator returns TRUE if the left operand lies within the range of the right operand. Otherwise, it returns FALSE. **Examples** 9 BETWEEN 1 AND 5 returns FALSE

Operator	Description
LIKE	Like. This binary operator takes a string value as its left operand and a wild-carded string value as its right operand. It returns TRUE if the left operand matches the right operand. Otherwise, it returns FALSE. You can use the wildcard '%' for any character sequence, and '_' for any single character. **Examples** • 'fred' LIKE 'f%' returns TRUE • 'fred' LIKE 'fr_d' returns TRUE **Note:** You can include a literal % or _ character by escaping it with a backslash.
NOT	Logical not. This unary operator takes a Boolean value as its operand and returns the inverse of the value. **Examples** • NOT TRUE returns FALSE • NOT (1 > 3) returns TRUE
AND	Logical and. This binary operator takes two Boolean operands. It returns TRUE if both operands are TRUE. Otherwise, it returns FALSE. **Examples** • TRUE AND TRUE returns TRUE • 1 < 2 AND 'fred' LIKE 'f%' returns TRUE
OR	Logical inclusive or. This binary operator takes two Boolean operands. It returns TRUE if either or both operands are TRUE. Otherwise, it returns FALSE. **Examples** • TRUE OR TRUE returns TRUE • (1 < 2) OR (3 = 4) returns TRUE

Mathematical operators

Mathematical operators usually combine numeric values to produce numeric results.

The two mathematical operators that do not follow this rule are the unary minus operator and brackets. The unary minus operator negates its operand. Brackets force the evaluation order of expressions.

Operator	Description
()	Use brackets to force the evaluation order of an expression. Brackets have the highest priority and are evaluated before other parts of the overall expression. Nested brackets are evaluated from the innermost to the outermost. **Examples** (2 + 4) / 3 = 6 / 3 = 2
&	The ampersand (&) operator is used to separate values when adding a list of values to an array. **Examples** `$ArrayVar := 'some text' & 123.45`
-	Unary minus. This operator negates its single operand. **Examples** -6
+	Addition. This binary operator takes two numeric operands and returns the sum of their values. **Examples** 2 + 4 = 6
-	Subtraction. This binary operator takes two numeric operands and returns the difference of their values. **Examples** 5 - 2 = 3
*	Multiplication. This binary operator takes two numeric operands and returns the product of their values. **Examples** 5 * 2 = 10
/	Division. This binary operator takes two numeric operands and returns the result of dividing the left operand's value by the right operand's value. **Examples** 10 / 2 = 5

Order of precedence for operators

Where an expression contains more than one operator, IBM Cognos Data Manager applies an order of precedence to determine the order in which it should perform evaluation.

In the following table, operators with higher priority numbers take precedence over those with lower priority numbers.

Where operators have the same priority number, Cognos Data Manager evaluates the expression from left to right.

If an expression includes nested brackets, Cognos Data Manager first evaluates the innermost brackets.

Priority	Operators
8	Brackets
7	Unary minus, that is, negation
6	* /
5	+ -
4	Logical comparisons, that is, !=, <>, >, >=, =, <=, <, IS, IN, LIKE, BETWEEN
3	NOT
2	AND
1	OR

Examples

The following examples show different levels of precedence:

- In this example, the brackets have priority 8, whereas / and * have priority 6.

  (10 / 2) * 5

  Following the order of precedence, IBM Cognos Data Manager first evaluates the expression within the brackets giving (5) * 5, and then applies the multiplication operator, giving 25.
- In this example, both operators, / and *, have priority 6.

  10 / 2 * 5

  Cognos Data Manager evaluates the expression from left to right, giving 10 / 2 * 5 = 5 * 5 = 25
- In this example, * has the highest priority.

  5 + 2 * 5 - 2

  Cognos Data Manager first evaluates the multiplication, giving 5 + 10 - 2. Both remaining operators have priority 5. Therefore, Cognos Data Manager evaluates the remaining expression from left to right, giving 5 + 10 - 2 = 15 - 2 = 13
- In this example, the brackets have the highest priority.

  (5 + 2) * (5 - 2)

  Cognos Data Manager first evaluates the expressions within the brackets, and then evaluates the remaining single operator (*), giving

  (5 + 2) * (5 - 2) = 7 * 3 = 21

Branch controls in scripts

The scripting language provides two-way branch control (IF statements) and multi-way branch control (CASE statements).

IF statements in scripts

IF statements provide two-way branch control. The branch that is taken depends upon the evaluation of a Boolean test expression.

The syntax for an IF statement is like this:

```
IF <expression> THEN
  <statement_block>;
[ELSE
  <statement_block>;]
```

If the test expression evaluates to TRUE, IBM Cognos Data Manager executes the statement block that follows the THEN keyword. Otherwise, if an ELSE clause is present, Cognos Data Manager executes the statement block of that clause.

This example illustrates the use of the IF construct. You can use derivations as the basis for conditional partitioning of data by fact deliveries.

```
IF( IntroductionDate < TODATE('01-01-2000','mm-dd-yyyy'))Then
  RETURN 'Legacy';
ELSE
  RETURN 'Current';
```

CASE statements in scripts

CASE statements provide multi-way branch control. The branch that is taken depends upon the evaluation of a test expression.

The result of the expression is compared to a list of literal values and the branch taken corresponds to the first match. If none of the literal values are matched, then the DEFAULT branch is taken. If none of the literal values are matched and no DEFAULT branch is specified, control passes to the statement that follows the CASE statement.

This example illustrates the use of the CASE construct with numeric literals. It converts a monetary amount to Euros.

```
CASE (CountryCode) OF
  BEGIN
    1  : $ConvRate := 6.55957;
    2  : $ConvRate := 1.95583;
    5  : $ConvRate := 13.7603;
    6  : $ConvRate := 1936.27;
    7  : $ConvRate := 2.20371;
   17  : $ConvRate := 40.3399;
   19  : $ConvRate := 166.386;
   22  : $ConvRate := 5.94573;
   DEFAULT :
     BEGIN
       $QrySpec := Concat('SELECT Rate FROM ConversionRate\
           WHERE CountryCode = ', ToChar(CountryCode));
       $ConvRate := Lookup('Sales', $QrySpec );
     END
  END
RETURN (Quantity*UnitPrice)/$ConvRate;
```

Where the country is part of the European Monetary Union, its currency has a constant exchange rate with the Euro. For these countries, you can hard-code the conversion rate, which is much faster than retrieval from a data table. In this fragment, these are countries with country code 1, 2, 5, 6, 7, 17, 19, and 22. Other countries have dynamic exchange rates, which are stored in the table ConversionRate. The default branch of the CASE statement uses the "Lookup" on page 26 function to retrieve this data for these countries.

When using alphabetic literals in a CASE statement, the literal value must be delimited by single quotation marks, as illustrated in this example.

```
CASE (SubStr( PRODUCT, 1, 1 )) OF
   BEGIN
    'B':$returnvar := 'Product name starting with B';
    'D':$returnvar := 'Product name starting with D';
    'F':$returnvar := 'Product name starting with F';
    'L':$returnvar := 'Product name starting with L';
    DEFAULT:$returnvar := 'Another product name';
   END
RETURN $returnvar;
```

IF or CASE

You can simulate CASE statements using nested IF statements. For example, in the preceding example, you could write the following:

```
IF (CountryCode = 1) THEN $ConvRate := 6.55957;
ELSE IF (CountryCode = 2) THEN $ConvRate := 1.95583;
ELSE IF (CountryCode = 5) THEN $ConvRate := 13.7603;
ELSE IF (CountryCode = 6) THEN $ConvRate := 1936.27;
ELSE IF (CountryCode = 7) THEN $ConvRate := 2.20371;
ELSE IF (CountryCode = 17) THEN $ConvRate := 40.3399;
ELSE IF (CountryCode = 19) THEN $ConvRate := 166.386;
ELSE IF (CountryCode = 22) THEN $ConvRate := 5.94573;
ELSE
  BEGIN
    $QrySpec := Concat('SELECT Rate FROM ConversionRate\
        WHERE CountryCode = ', ToChar(CountryCode));
    $ConvRate := Lookup('Sales', $QrySpec );
  END
RETURN (Quantity*UnitPrice)/$ConvRate;
```

Using IF statements this way requires evaluation of each test expression until the correct branch is identified. This can be less efficient than a CASE statement, which requires the evaluation of only one expression. However, you must use nested IF statements where one expression cannot determine which branch to take. For example:

```
IF (VendorID = 12) THEN $ConvRate := 5.99723;
ELSE IF (CountryCode = 1) THEN $ConvRate := 6.55957;
```

Loops in scripts

The scripting language provides the WHILE statement which is a looping construct. It uses a Boolean expression to determine whether to execute a block of statements.

The block of statements is executed repeatedly until the test expression evaluates to FALSE.

The WHILE statement has this syntax:

```
WHILE <expression> DO
 BEGIN
  <statement_list>;
 END
```

The following example illustrates the use of the WHILE construct and statement blocks. It iterates until a particular text file exists and can be read. To avoid excessive disk activity, this example pauses for ten seconds before rechecking for the text file.

```
WHILE (NOT FileCheck( 'd:\\temp\\test.txt', 'READ') )
DO
  BEGIN
    LogMsg('Waiting for file test.txt');
    Delay(10);
  END
```

The WHILE loop is a preconditioned construct. However, it is possible to simulate postconditioned loops. To do this, either initialize variables to ensure that the loop is entered, or repeat the conditional block of statements before the WHILE construct. For example

```
WHILE ($TestVar > 0) DO
  BEGIN
    TestMsg := Concat('TestVar = ', ToChar($TestVar));
    LogMsg($TestMsg);
    $TestVar := $TestVar - 1;
  END
```

To convert this to simulate a postconditioned loop, repeat the conditional block of statements like this:

```
TestMsg := Concat('TestVar = ', ToChar($TestVar));
LogMsg($TestMsg);
$TestVar := $TestVar - 1;
WHILE ($TestVar > 0) DO
  BEGIN
    $TestMsg := Concat('TestVar = ', ToChar($TestVar));
    LogMsg($TestMsg);
    $TestVar := $TestVar - 1;
  END
```

Nested scripts

You can nest scripts by calling a user-defined function in a script. You can also base a derivation on another derivation.

When a script calls a user-defined function, the user-defined function must be fully defined. That is, all formal parameters and all variables local to the function must be defined.

A derivation must appear after any derivation from which it is derived.

Variables in scripts

IBM Cognos Data Manager supports two types of variables that you can use with the scripting language.

The variable types are as follows:

- Property variables, which are intrinsic to Cognos Data Manager and affect the operation of Cognos Data Manager commands.

- User-defined variables, which store values for use in your scripts.

You can use variables in the following instances:

- As normal variables in which you can store values and from which you can retrieve values during the execution of a script.
- As substitution variables, in which Cognos Data Manager replaces the variable with the value it contains.

For more information, see Variables in the IBM Cognos Data Manager *User Guide*.

Referring to variables in scripts

To assign a value to a variable or to access the value that a variable contains, use the name of the variable prefixed with a dollar symbol.

For example, use $TextVar to refer to that variable within a script:

```
$TextVar := 'This is a text variable';
LogMsg($TextVar);
```

Data types in variables

There are a number of data types are available for variables.

ARRAY

Variables of type ARRAY can store an array of an unlimited number of values of mixed data types.

The index of the first value is 1. You can assign the following value types to ARRAY variables:

- One literal value of any (non-array) data type. For example:

  ```
  $ArrayVar := 'some text',
  and $ArrayVar :=
  123.45
  ```

- A set of literal values, with ampersands (&) separating the values in the set. For example :

  ```
  $ArrayVar := 'some text'
  & 123.45
  ```

- Another array. For example:

  ```
  $ArrayVar_1 := $ArrayVar_2
  ```

- A set of variables, which may include arrays. For example:

  ```
  $ArrayVar := $DateVar & $TextVar & $ArrayV2
  ```

- A set of variables and literal values. For example:

  ```
  $ArrayVar := 'some text' & $IntVar
  ```

When you add to an array that already has a list of items, the position of the new value determines its position within the array. In the following example, the new value is added to the end of the list:

```
$ArrayVar := $ArrayVar & 'some text'
```

This example adds the new value to the beginning of the list:

```
$ArrayVar := 'some text' & $ArrayVar
```

Notes

- If you assign an array to a value of another data type, IBM Cognos Data Manager uses the first value in the array. Type conversion rules are the same as if you had assigned this value directly. For example, in the following fragment 'some text' is assigned to the TextVar variable.

```
$ArrayVar
:= 'some text' & 123.45;
$TextVar := $ArrayVar;
```

- User-defined functions can have no more than 16 parameters. However, you can use arrays to pass more than this number of values to a user-defined function.
- User-defined functions can return one result. However, you can use an array to return many values.

BOOLEAN

A Boolean value. Use the string literal value 'TRUE' or non-zero values to represent TRUE; use 'FALSE' or zero to represent FALSE.

Variables of type BOOLEAN can store the Boolean values TRUE and FALSE. You can assign the following value types to BOOLEAN variables:

- The literal string value 'TRUE', in upper, lower, or mixed case, to represent the value TRUE or any other string value to represent FALSE.
- Numeric values where non-zero values represent TRUE and zero represents FALSE.
- The result of a logical comparison. The following example assigns FALSE to the variable BoolVar

```
$BoolVar := ('one' = 'two')
```

- The value of another variable of type BOOLEAN. For example:

```
$BoolVar2 := $BoolVar1
```

- The result of casting a value to a numeric or string value. In this case, 'TRUE' and non-zero numbers represent TRUE and any other values represent FALSE.

BINARY

Variables of type BINARY can store binary values.

BINARY variables have a precision, that is, BINARY(<p>), and the name of the function to show that the data is missing from the text, this should be tohex().

You can assign the following value types to BINARY variables:

- The value of another variable of type BINARY. For example:

```
$BinaryVar2 := $BinaryVar1
```

The “ToHex” on page 6 function can be used to show binary data in hexadecimal format.

CHAR(<n>)

Variables of type CHAR can store a minimum of one character, and a maximum of 8000 characters.

However, you can specify a number, for example CHAR(255) specifies that the variable can store up to 255 characters. You can assign the following value types to CHAR variables:

- String literal values delimited by single quotation marks. For example:

```
$TextVar := 'variable'
```

- A string literal value may contain embedded single quotation marks, but you must escape these with an additional single quotation mark. For example, to assign the literal value, "variable's contents" to the variable TextVar, use the following:

  ```
  $TextVar := 'variable''s contents'
  ```

- The value of another variable of type CHAR. For example:

  ```
  $TextVar2 := $TextVar1
  ```

- The result of using the “ToChar” on page 3 function to change a value to a string representation of that value. The following example assigns '123' to TextVar if IntVar = 123

  ```
  $TextVar := ToChar($IntVar)
  ```

Tip: Values in double quotation marks are identifiers and produce a runtime error if used in the wrong context, for example UDFs. An example where it works is in a derivation element where an identifier is the name of another model element, for example, a measure element.

DATE

Variables of type DATE can store date and timestamp (date/time) values. You can assign the following value types to DATE variables:

- The value of another variable of type DATE. For example:

  ```
  $DateVar2 := $DateVar1
  ```

- The result of using the “ToDate” on page 4 function to cast a value to a date or timestamp. In the following example the date December 15 2001 is assigned to the variable DateVar

  ```
  $DateVar := ToDate('12-15-2001', 'mm-dd-yyyy')
  ```

DATE WITH TIMEZONE

Variables of type DATE WITH TIME ZONE can store date and timestamp (date/time) within a time zone values.

You can assign the following value types to DATE WITH TIME ZONE variables:

- The value of another variable of type DATE WITH TIME ZONE. For example:

  ```
  $DateTZVar2 := $DateTZVar1
  ```

- The value of a variable of type DATE. For example:

  ```
  $DateTZVar2
  := $DateVar1
  ```

- The result of using the function to cast a value to a date or timestamp. In the following example, the date December 15 2001 EST is assigned to the variable:

  ```
  DateTZVar$DateTZVar := ToDate('12-15-2001
  -05:00', 'mm-dd-yyyy stzh:tzm')
  ```

FLOAT

Variables of type FLOAT can store a double-precision, floating point number.

You can assign the following value types to FLOAT variables:

- Floating point or integer literal values. For example:

  ```
  $FloatVar := 123.45
  ```

- The value of another variable of type FLOAT. For example:

  ```
  $FloatVar2 := $FloatVar1
  ```

- The result of a mathematical expression. For example:

  ```
  $FloatVar := (123.4*2.3)/5.
  ```

- The result of using the "ToDouble" on page 4 function to cast a value to a floating point number. In the following example 123.45 is assigned to FloatVar if TextVar = '123.45'

  ```
  $FloatVar := ToDouble($TextVar)
  ```

- The result of using the "ToInteger" on page 6 function to cast a value to an integer number. For example:

  ```
  $FloatVar
  := ToInteger('123').
  ```

INTEGER

Variables of type INTEGER can store an integer value within the range permitted by your system.

You can assign the following value types to INTEGER variables:

- Integer literal values. For example:

  ```
  $IntVar := 12345
  ```

- The value of another variable of type INTEGER. For example:

  ```
  $IntVar2 := $IntVar1
  ```

- The result of a mathematical expression. For example:

  ```
  $IntVar := 123*4
  ```

- The result of using the "ToInteger" on page 6 function to cast a value to an integer. In the following example, 123 is assigned to IntVar if TextVar = '123'

  ```
  $IntVar := ToInteger($TextVar)
  ```

INTERVAL DAY TO SECOND

Variables of type INTERVAL DAY TO SECOND can store INTERVAL DAY TO SECOND values.

You can assign the following value types to INTERVAL DAY TO SECOND variables:

- The value of another variable of type INTERVAL DAY TO SECOND. For example:

  ```
  $IntDSVar2 := $IntDSVar1
  ```

- The result of using the "ToIntervalDS" on page 7 function to cast a value to INTERVAL DAY TO SECOND. In the following example, the day to second interval '001 12:13:14' is assigned to the variable IntDSVar

  ```
  $IntDSVar := ToIntervalDS('01 12:13:14',
  ' dd hh:mm:ss')
  ```

INTERVAL YEAR TO MONTH

Variables of type INTERVAL YEAR TO MONTH can store INTERVAL YEAR TO MONTH values.

You can assign the following value types to INTERVAL YEAR TO MONTH variables:

- The value of another variable of type INTERVAL YEAR TO MONTH. For example:

  ```
  $IntYMVar2 := $IntYMVar1
  ```

- The result of using the "ToIntervalYM" on page 8 function to cast a value to INTERVAL YEAR TO MONTH. In the following example, the year to month interval '010-10' is assigned to the variable IntYMVar

  ```
  $IntYMVar := ToIntervalYM('010-10', 'yyyy-mm')
  ```

NUMBER(<p>,<s>)

A precise number with precision <p> and scale <s>.

Variables of type NUMBER can store precise numbers with a maximum precision of 77 significant figures. You can specify the precision and the number of digits to the right of the decimal point (the scale). For example, NUMBER(10,4) specifies a precise number with precision of 10 significant figures and a scale or 4. You can assign the following value types to NUMBER variables:

- Floating point or integer literal values. For example:

  ```
  $NumVar := 123.45
  ```

- The value of another variable of type NUMBER. For example:

  ```
  $NumVar2 := $NumVar1
  ```

- The result of a mathematical expression. For example:

  ```
  $NumVar := (123.4*2.3)/5
  ```

- The result of using the “ToDouble” on page 4 or “ToInteger” on page 6 functions to cast a value to a floating point number or integer respectively.

TIME

Variables of type TIME can store time values.

You can assign the following value types to TIME variables:

- The value of another variable of type TIME. For example:

  ```
  $TimeVar2 := $TimeVar1
  ```

- The result of using the “ToTime” on page 9 function to cast a value to a time. In the following example, the Time '12:13:14' is assigned to the variable TimeVar

  ```
  $TimeVar := ToTime('12:13:14', 'hh:mm:ss')
  ```

TIME WITH TIMEZONE

Variables of type TIME WITH TIME ZONE can store time within a time zone values.

You can assign the following value types to TIME WITH TIME ZONE variables:

- The value of another variable of type TIME WITH TIME ZONE. For example:

  ```
  $TimeTZVar2 := $TimeTZVar1
  ```

- The value of a variable of type TIME. For example:

  ```
  $TimeTZVar2
  := $TimeVar1
  ```

- The result of using the function to cast a value to a time. In the following example, the Time '12:13:14 PST' is assigned to the variable TimeTZVar

  ```
  $TimeTZVar := ToTime('12:13:14
  -08:00', 'hh:mm:ss stzh:tzm')
  ```

Substitution variables

Variable substitution is the process in which IBM Cognos Data Manager replaces a variable with a specific value.

This process occurs when the script or definition is parsed and remains in effect until it is next parsed.

Because variable substitution occurs before the script is executed, you cannot assign a value to a substituted variable. That is, a substitution variable provides a

constant value during the execution of the object in which the variable is declared. You must ensure that such variables are assigned suitable values before the script is called.

Variable substitution evaluates a variable once during execution of the build or JobStream in which the variable is declared. Without variable substitution, Cognos Data Manager evaluates the variable each time it is referenced. This method can be less efficient, especially for derivation elements, where the variable is evaluated for every data row.

You can use substitution variables in Cognos Data Manager definitions and scripts, including in the definition of derivation elements. You can use non-substituted variables only within expressions and scripting statements.

To indicate that Cognos Data Manager should perform substitution on a variable, enclose the variable in braces. For example, use {$Example} to specify that Cognos Data Manager should perform substitution on the variable named Example.

Examples

- `$QrySpec := 'SELECT * FROM Customer WHERE Country=''{$EXAMPLE}'';'`

  If the variable EXAMPLE contains 'France', performing substitution on the statement results in the following:

  ```
  $QrySpec := 'SELECT * FROM Customer WHERE Country=''France'';'
  ```

  Cognos Data Manager only refers to the variable as it parses the script or definition. It does not refer to the variable during execution of the script. Therefore, changes in the value of EXAMPLE during execution of the script have no effect.

- `IF ({$s1} = {$s2}) THEN RETURN {$S3}; ELSE RETURN {$S4};`

  If variables S1, S2, S3, and S4 are NULL when the script is parsed, substitution on this script produces the following:

  ```
  IF ( = )
  THEN RETURN;
  ELSE RETURN;
  ```

  This is invalid and produces a runtime error, even though the script is valid without substitution.

Quotation marks

When you perform variable substitution, the value of that variable replaces the name of the variable when the script or definition is parsed.

From that point, IBM Cognos Data Manager interprets the substituted text as if you entered it manually. Therefore, where the substituted variable is a string parameter, you should enclose it in single quotation marks.

Examples

Assume that the variable QrySpec contains the SQL statement, "SELECT ProductName FROM Product WHERE ProductNumber=1". Without substitution, the following statement assigns the name of the product with ProductNumber=1 to the RESULT variable.

```
$RESULT := Lookup( 'GO_Sales', $QrySpec );
```

(Strictly, it assigns a comma-separated list having one item). However, if you use substitution, the substituted variable may require quotation marks. Consider this statement:

```
$RESULT := Lookup( 'GO_Sales', {$QrySpec} );
```

During substitution, it is replaced with the following statement:

```
$RESULT := Lookup( 'GO_Sales', SELECT ProductName FROM Product
WHERE
ProductNumber=1 );
```

Because there are no delimiting single quotation marks around the SELECT statement, this results in a parse error. To avoid this problem, you must enclose the substitution in single quotation marks, like this:

```
$RESULT := Lookup( 'GO_Sales', '{$QrySpec}' );
```

Text that contains quotation marks:

Where text contains single quotation marks, you may need to escape those quotation marks.

For example, assume that the variable QrySpec contains this SQL statement:

```
SELECT CustName FROM Customer WHERE Country='France'
```

Consider variable substitution on the following statement:

```
$RESULT := Lookup( 'GO_Sales', '{$QrySpec}' );
```

After substitution it becomes the following statement:

```
$RESULT := Lookup( 'GO_Sales', 'SELECT CustName FROM Customer
WHERE Country='France'' );
```

This is syntactically incorrect because the single quotation mark before "France" terminates the statement. To avoid this, you must add a second single quotation mark to each quotation mark that is embedded in the contents of a substituted variable. In the preceding example, QrySpec should contain the following:

```
SELECT CustName FROM Customer WHERE Country=''France''
```

Script syntax

```
<statement_list>
 <statement_list> ::=
 <statement>;{<statement>;}
<statement_block> ::=
 <statement>;|BEGIN <statement_list> END
<statement> ::=
 <if_statement>|case_statement>|assignment>|
 <while_statement>|return_statement>|<function_call>
```

Symbol	Description
<statement_list>	A sequence of statements in the IBM Cognos Data Manager scripting language.
<statement>	A single statement.

Symbol	Description

<statement_block>	A statement list that is enclosed in the BEGIN and END keywords. For example:
	`BEGIN`
	`  LogMsg('Procedure Successful');`
	`  $EXAMPLE := TRUE;`
	`END`
	If the statement list contains only one statement, then the use of the BEGIN and END keywords is optional.

<if_statement>	A construct that conditionally executes a block of statements depending on the value of a Boolean expression. If the expression evaluates to TRUE, then Cognos Data Manager executes the statement block that follows the THEN keyword, otherwise, it executes the statement block (if any) that follows the ELSE keyword.

<case_statement>	A construct that conditionally executes one of several statement blocks depending on the value of an expression. Cognos Data Manager evaluates the expression and then, starting from the top, compares the result with each literal value in turn. If the result of the expression matches the literal value, then Cognos Data Manager executes the corresponding statement block. If the result of evaluating the expression matches none of the literal values, then Cognos Data Manager executes the DEFAULT clause if it exists. If no literal values match the expression and there is no DEFAULT clause, execution continues with the statement that follows the END keyword.
	For example:
	`CASE $VAR OF`
	`BEGIN`
	`  1 : LogMsg('Result is 1');`
	`  2 : LogMsg('Result is 2');`
	`  DEFAULT : LogMsg('Result is not 1 or 2');`
	`END`

<assignment>	A statement that assigns to a variable the result of executing an expression. **Note:** This uses a Pascal-like assignment operator (:=).
<while_statement>	A preconditioned control structure that executes a statement block while a logical expression evaluates to TRUE.
<return_statement>	A statement that ends execution of the script and returns the value of the specified expression to the calling process.
<function_call>	A call to a built-in function or user-defined function.

```
<if_statement> ::=
 IF <logical_expression> THEN <statement_block>
 ELSE <statement_block>]
```

Symbol	Description
<if_statement>	A construct that conditionally executes a block of statements depending on the value of a Boolean expression.
IF	This keyword introduces an IF statement.
<logical_expression>	The Boolean expression that Cognos Data Manager evaluates to determine which (if any) statement block to execute.
THEN	This keyword introduces the statement block that Cognos Data Manager executes if the Boolean expression evaluates to TRUE.
<statement_block>	A block of statements.
ELSE	If present, this keyword introduces the statement block that Cognos Data Manager executes if the Boolean expression evaluates to FALSE.

```
<logical_expression> ::=
 [[<expression>] <logical_operator>] <expression>|
 (<logical_expression>)
<logical_operator> ::=
 AND|OR|NOT|!|=|!=|<|>|<=|>=
```

Symbol	Description
<logical_expression>	An expression that can evaluate only to either TRUE or FALSE.
<expression>	A literal value or an expression that evaluates to a value.
<logical_operator>	A Boolean operator. For information on the available logical operators, see "Logical operators" on page 83. For information on how Cognos Data Manager compares values, see "Comparison of values in scripts" on page 82.

```
<expression> ::=
 <value>|<logical_expression>|<arith_expression>|
 <function_call>|(<expression>)
<value> ::=
 <variable>|<literal>
```

Symbol	Description
<expression>	A literal value or an expression that evaluates to a value.
<value>	Either a literal value or the value that a variable contains.

Symbol	Description
<logical_expression>	An expression that can evaluate only to either TRUE or FALSE.
<arith_expression>	An arithmetic expression. That is, an expression that results in a numeric value.
<function_call>	A call to a built-in function or user-defined function.
<variable>	A variable that stores a discrete value.
<literal>	A discrete, literal value.

```
<arith_expression> ::=
 [[<expression>] <arith_operator>  <expression>|
 (<arith_expression>)
<arith_operator> ::=
 +|-|*|/
```

Symbol	Description
<arith_expression>	An arithmetic expression. That is, an expression that results in a numeric value.
<expression>	A literal value or an expression that evaluates to a value.
<arith_operator>	A mathematical operator. For information on the available logical operators, see "Logical operators" on page 83. For information on how Cognos Data Manager compares values, see "Comparison of values in scripts" on page 82.

```
<function_call> ::=
 <function_name>([<value>]{, <value>})
```

Symbol	Description
<function_call>	A call to a built-in function or user-defined function.
<function_name>	The name of the function that the line of syntax calls.
<value>	An actual parameter of the function.

```
<case_statement> ::=
 CASE <expression> OF
 BEGIN
  <case_list>
 END
```

```
<case_list> ::=
 <literal>: <statement_block> {<literal>: <statement_block>}
 [DEFAULT: <statement_block>]
```

Symbol	Description
<case_statement>	A construct that conditionally executes one of several statement blocks depending on the value of an expression.
<expression>	A literal value or an expression that evaluates to a value.
<case_list>	A series of literal values, each of which has a corresponding statement block. Cognos Data Manager compares the result of <expression> with each literal value in turn, and executes the statement block that corresponds to the first literal value that matches the result.
<literal>	A discrete, literal value.
<statement_block>	A block of statements.
DEFAULT	This keyword introduces the statement block that Cognos Data Manager executes if <expression> matches none of the literal values of the case list.

```
<while_statement> ::=
 WHILE <expression> DO <statement_block>
```

Symbol	Description
<while_statement>	A preconditioned control structure that executes a statement block while a logical expression evaluates to TRUE.
<expression>	The expression that Cognos Data Manager tests to determine whether to execute the statement block.
<statement_block>	The block of statements that Cognos Data Manager executes if <expression> evaluates to TRUE.

```
<return_statement> ::=
 RETURN <expression>
```

Symbol	Description
<return_statement>	A statement that ends execution of the script and returns the value of the specified expression to the calling process.

Symbol	Description
<expression>	An expression that evaluates to the value that Cognos Data Manager returns to the calling process.

```
<assignment> ::=
 <variable> := <expression>
```

Symbol	Description
<assignment>	A statement that assigns to a variable the result of evaluating an expression. **Note:** This uses a Pascal-like assignment operator (:=).
<variable>	The variable to which Cognos Data Manager assigns the result.
<expression>	A discrete, literal value or an expression that evaluates to a discrete value.

Debugging scripts

To debug IBM Cognos Data Manager scripts, you should insert statements that write to the build or JobStream execution log. By doing this, you can track the logic flow through the script and obtain the values of expressions and variables as execution of the script progresses.

For example

```
$TestVar := 'This is a test line for the log';
LogMsg(Concat('$TestVar = ', $TestVar));
```

This results in a message, in the execution log, similar to this

```
[USER     - 14:51:35] $TestVar = This is a test line for the log
```

When you execute a fact build, dimension build, or JobStream it is possible to create an additional log file that traces any expressions and scripts, user-defined functions, and Cognos Data Manager predefined functions. This is useful when a lower level of debugging is required for complex transformation logic.

For more information, see Create a Log File for Expression and Script Tracing in the IBM Cognos Data Manager *User Guide*.

Activate debugging in scripts

Each message in an IBM Cognos Data Manager execution log has an associated group. For example, messages that relate to memory usage belong to the INTERNAL group.

All messages that arise from the "LogMsg" on page 26 function belong to the USER group.

You must elect to write messages for the USER group to use the LogMsg function for debugging. You can do this in Cognos Data Manager Designer, on the command line, or using an environment variable.

Activating debugging using IBM Cognos Data Manager Designer

This topic describes how to activate debugging in scripts using IBM Cognos Data Manager Designer.

Procedure

1. At execution time, from the **Execute Build** or **Execute JobStream** dialog box, select **Override Build Settings** (or **Job/JobStream Settings**), and then click **User**.
2. From the **Fact Build**, **Dimension Build**, or **JobStream Properties** window, click the **Logging** tab, and from the **Trace** box, click **User**.

Activating debugging using the command line

This topic describes how to activate debugging in scripts using the command line.

Procedure

Include USER in the TRACE_VALUES command line parameter.
For example, to cause USER and PROGRESS messages to be written to the log, use -VTRACE_VALUES=USER,PROGRESS

Activating debugging using an environment variable

This topic describes how to activate debugging in scripts using an environment variable.

Procedure

Set the environment variable TRACE_VALUES to include USER.

Note: In Windows, you must set the TRACE_VALUES environment variable before starting IBM Cognos Data Manager.

Conditionally write debug messages

You can control whether IBM Cognos Data Manager writes debug messages to the execution log by choosing whether to write USER messages to the log.

However, not choosing to write USER messages to the log may not be practical perhaps because you want USER messages other than debug messages. Additionally, if a user selects the USER group, all your debug messages are written to the execution log. We suggest that you use a variable to control whether debugging messages are written to the execution log, and suggest the following methods:

- "Simple debugging"
- "Hierarchical debug categories" on page 104
- "Multiple debug categories" on page 104

Simple debugging

In this method, you can switch debugging on or off.

You declare a variable of type BOOLEAN that you initialize to TRUE to switch debugging on and FALSE to switch debugging off. Make all debug messages conditional on this variable.

Note: You must also elect to write USER messages to the execution log.

Examples

- In the build or JobStream definition, include this declaration:

  ```
  DECLARE 'DEBUG' BOOLEAN 'TRUE'
  ```

- In the script, include code similar to this:

  ```
  IF ($DEBUG) THEN  LogMsg ('This is a debug message');
  ```

To turn off debugging in this method, change the initialization of DEBUG to 'FALSE'.

Hierarchical debug categories

In this method, you declare an integer variable that you initialize to indicate the level of debugging that you require.

In the script, include either case statements or nested IF statements to write debug messages to the execution log. As with all other debugging, you must also elect to write USER messages to the log.

Examples:

In the build or JobStream definition, declare a variable of type INTEGER, initialized to indicate the level of debugging that you require.

- ```
  DECLARE 'DEBUG_LEVEL' INTEGER 3
  ```

  In the script, include case statements:

  ```
  CASE ($DEBUG_LEVEL)
  OF
    BEGIN
      1 : LogMsg('Debug level is 1');
      2 : LogMsg('Debug level is 2');
      3 : LogMsg('Debug level is 3');
    END
  ```

  This example performs different actions depending on the value of the DEBUG_LEVEL variable. You must define messages for each value of this variable for which you want a message in the execution log.

- ```
  DECLARE 'DEBUG_LEVEL' INTEGER 3
  ```

  In the script, use nested IF statements to write messages to the log:

  ```
  IF ($DEBUG_LEVEL >=
  3) THEN
    LogMsg('Debug level is at least 3');
  ELSE IF ($DEBUG_LEVEL >= 1) THEN
    LogMsg('Debug level is at least 1');
  ```

To turn off debugging in this method, set the DEBUG_LEVEL variable to zero.

Multiple debug categories

In this method, you can create individual categories and assign each debug message to one or more of those categories.

Using the "InStr" on page 56 function, you can test whether the category of each message is to be included in the execution log.

Examples:

In the build or JobStream definition, declare a variable of type CHAR, initialized to include all debug categories you want to include in the execution log:

```
DECLARE 'DEBUG_CAT' CHAR(32) 'CAT_1,CAT_2'
```

For each message, test whether the category is to be written to the log:

```
IF (InStr($DEBUG_CAT, 'CAT_1')) THEN
  LogMsg('Debug Category 1');
```

To turn off debugging in this method, set the DEBUG_CAT variable to an empty string.

Hints and tips when creating scripts

The following topics contain information that may assist you when you are creating scripts:

- create functions from derivations
- expressions or scripts
- functions to initialize variables in scripts
- initializing variables from a data table in scripts

Create functions from derivations

Where a derivation from one fact build has potential for use in other fact builds, you should consider converting that derivation to a user-defined function so that you can reuse your work. When you convert a derivation in this way, consider the following points:

- You cannot use variable substitution in a user-defined function. Replace substitution variables with function arguments.
- The names of elements may change from fact build to fact build. You should replace transformation model elements with function arguments.

Examples

Assume that you have a derivation element that calculates the gross profit of a sale using the formula Quantity*(UnitSalePrice-Unitcost). Here, Quantity, UnitSalePrice, and Unitcost are the names of transformation model elements. To convert this derivation to a user-defined function, you must create three parameters, one for each element. If this function is named GrossProfit, it will have a usage syntax similar to GrossProfit(<quantity>, <unitsaleprice>, <unitcost>).

Expressions or scripts

Wherever an object must return a value (for example, derivations, user-defined functions, and filters), you can use simple expressions in place of scripts.

IBM Cognos Data Manager evaluates the expression and returns the result. When deciding whether to use expressions or scripts, you should consider the following:

- Scripts must use RETURN statements to return values; expressions implicitly return values.
- Scripts can be of unlimited length and complexity; expressions are limited to one logical line.

Note: You cannot use expressions where the object may not return a value (for example, JobStream procedures).

Functions to initialize variables in scripts

Where a variable exists in many builds and has the same initial expression for each build, consider creating a user-defined function to initialize the variable.

You can then reuse the initialization routine, which is particularly useful where the initialization of the variable is complex.

Initializing variables from a data table in scripts

If initialization values of variables change frequently, or are dependent on the results of another build, you can create a variable of the appropriate type and set the initial expression appropriately.

For example, create a "TextVar" variable with a suitable CHAR precision and set the initial expression to

```
DECLARE TextVar CHAR(32) LOOKUP('myConn',\
'SELECT InitValue FROM Init WHERE VarName=''TextVar''')
```

An additional benefit of this approach is that you can change the initial value of a variable without running IBM Cognos Data Manager Designer or editing a text definition.

Notices

This information was developed for products and services offered worldwide.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. This document may describe products, services, or features that are not included in the Program or license entitlement that you have purchased.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Software Group
Attention: Licensing
3755 Riverside Dr
Ottawa, ON K1V 1B7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at " Copyright and trademark information " at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

G

I

L

M

N

O

P

Q

R

S

T

U

V

W