Table of Contents

Introduction 7

Chapter 1: Function Reference 9
  Conversion Functions 9
    SetTimeZone 10
    ToChar 12
    ToDate 12
    ToDouble 13
    ToHex 14
    ToInteger 15
    ToIntervalDS 16
    ToIntervalYM 17
    ToNumber 17
    ToTime 18
    ToTimeZone 19
  Control Functions 22
    ArrayAddItem 24
    ArrayClear 24
    ArrayDeleteItem 25
    ArrayItem 25
    ArrayModifyItem 26
    ArraySearch 26
    ArraySize 27
    ArraySort 27
    Audit 27
    AuditTrail 28
    DBMS 31
    DBName 32
    Delay 32
    Driver 33
    Exit 33
    FileCheck 34
    FileClose 35
    FileFromParts 35
    FileFullPath 36
    FileList 36
    FileOpen 37
    FileRead 37
    FileWrite 38
    GetDirectory 38
    GetFileName 39
    LogMsg 39
    Lookup 40
    MessageCode 40
Table of Contents

MessageCount 41
MessageSeverity 41
MessageText 42
NodeAuditID 42
NodeStatus 43
OpSys 43
RowNum 43
RowsInserted 43
RowsUpdated 44
SendAlert 45
SendMail 45
Sql 46
System 47
UUID 47
VariableInfo 48
SQL Cursor Functions 49
   Examples 49
   SQLPrepare 50
   SQLGetLastError 51
   SQLColumnCount 51
   SQLColumnName 51
   SQLColumnNo 52
   SQLBind 53
   SQLFetch 53
   SQLData 54
   SQLClose 55
Logical Functions 55
   Choose 55
   If 56
   IfNull 56
Mathematical Functions 56
   Abs 57
   Band 57
   Ceil 58
   Cos 58
   Exp 59
   Floor 59
   Ln 59
   Log 60
   Mod 60
   Power 61
   Rand 61
   Round 62
   Sign 62
   Sin 63
   Sqrt 63
   Tan 63
   Trunc 64
Member Functions 64

4 IBM Cognos Data Manager
IsAncestor 65
Level 65
Member 66
TypeInfo 66
Unmatched 67

Text Functions 68
Char 69
Checksum 69
Collapse 70
Concat 70
ConcatSep 71
CountStr 71
ExtractStr 71
I18NConvert 72
I18NString 73
Initcap 73
InStr 74
IsAlpha 75
IsAlphaNumeric 75
IsDigit 76
IsFloat 76
IsInteger 77
IsLower 77
IsNumeric 78
IsUpper 78
Left 79
Length 79
Lower 80
LPad 80
LTrim 81
Replace 81
Right 82
RPad 82
RTrim 83
Soundex 83
SubStr 84
Translate 85
Trim 85
Upper 86

Date Functions 86
AddDaysToInterval 87
AddMonthsToDate 88
AddMonthsToInterval 89
AddSecondsToInterval 89
AddToDate 90
AddYearsToDate 90
AddYearsToInterval 91
DaysBetween 92
FirstOfMonth 92
Chapter 2: The IBM Cognos Data Manager Scripting Language 101

Assignment Operator in Scripts 101
Returned Value in Scripts 102
Comparison of Values in Scripts 102
Numeric Values 102
Dates and Times 102
Characters and Strings 102
NULL Values 102
Operators 103
Logical Operators 103
Mathematical Operators 106
Order of Precedence for Operators 107
Branch Controls in Scripts 108
IF Statements in Scripts 109
CASE Statements in Scripts 109
Loops in Scripts 110
Nested Scripts 111
Variables in Scripts 111
Referring to Variables in Scripts 112
Data Types in Variables 112
Substitution Variables 117
Script Syntax 119
Debugging Scripts 124
Activate Debugging in Scripts 124
Conditionally Write Debug Messages 125
Hints and Tips when Creating Scripts 126
Create Functions from Derivations 127
Expressions or Scripts 127
Functions to Initialize Variables in Scripts 127
Initializing Variables from a Data Table in Scripts 127

Index 129
Introduction

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

The examples in this document use Bakus-Naur Form (BNF) to describe the syntax of the Data Manager language. The BNF syntax consists of the following:

- 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 placeholders. They 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.
- A set of rules that you apply when interpreting the BNF definitions:
  - ::= 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 piping 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:

```plaintext
<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 at http://publib.boulder.ibm.com/infocenter/cogic/v1r0m0/index.jsp. Updates to Release Notes are published directly to Information Centers.

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**

This product does not currently support accessibility features that help users with a physical disability, such as restricted mobility or limited vision, to use this product.
Chapter 1: Function Reference

IBM® Cognos® Data Manager provides predefined functions and operators that you can use, with the 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.

Data Manager provides the following function types:

- conversion
- control
- logical
- mathematical
- member
- text
- date
- SQL cursor

Conversion Functions

Conversion functions convert data from one data type to another.

These are the conversion functions:

- SetTimeZone
- ToChar
- ToDate
- ToDouble
- ToHex
- ToInteger
- ToIntervalDS
- ToIntervalYM
- ToNumber
- ToTime
- ToTimeZone
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.

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>The time zone. Possible values are</td>
</tr>
<tr>
<td></td>
<td>• 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</td>
</tr>
<tr>
<td></td>
<td>• UTC is Coordinated Universal Time (previously GMT)</td>
</tr>
<tr>
<td></td>
<td>• GMT is Greenwich Mean Time (superseded by UTC)</td>
</tr>
<tr>
<td></td>
<td>• ACDT is Australian Central Daylight Time (UTC+10:30)</td>
</tr>
<tr>
<td></td>
<td>• ACST is Australian Central Standard Time (UTC+9:30)</td>
</tr>
<tr>
<td></td>
<td>• AEDT is Australian Eastern Daylight Time (UTC+11:00)</td>
</tr>
<tr>
<td></td>
<td>• AEST is Australian Eastern Standard Time (UTC+10:00)</td>
</tr>
<tr>
<td></td>
<td>• AKDT is Alaska Daylight Time (UTC-8:00)</td>
</tr>
<tr>
<td></td>
<td>• AKST is Alaska Standard Time (UTC-9:00)</td>
</tr>
<tr>
<td></td>
<td>• ADT is Atlantic daylight Time (UTC-3:00)</td>
</tr>
<tr>
<td></td>
<td>• AST is Atlantic Standard Time (UTC-4:00)</td>
</tr>
<tr>
<td></td>
<td>• AWST is Australian Western Standard Time (UTC+8:00)</td>
</tr>
<tr>
<td></td>
<td>• BST is British Summer Time (UTC+1:00)</td>
</tr>
<tr>
<td></td>
<td>• CDT is Central Daylight Time (UTC-5:00)</td>
</tr>
<tr>
<td></td>
<td>• CST is (US) Central Standard Time (UTC-6:00)</td>
</tr>
<tr>
<td></td>
<td>• CEDT is Central European Daylight Time (UTC+2:00)</td>
</tr>
<tr>
<td></td>
<td>• CEST is Central European Summer Time (UTC+2:00)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>CET</td>
<td>Central European Time (UTC+1:00)</td>
</tr>
<tr>
<td>EDT</td>
<td>Eastern Daylight Time (UTC-4:00)</td>
</tr>
<tr>
<td>EST</td>
<td>(US) Eastern Standard Time (UTC-5:00)</td>
</tr>
<tr>
<td>EEDT</td>
<td>Eastern European Daylight Time (UTC+3:00)</td>
</tr>
<tr>
<td>EEST</td>
<td>Eastern European Summer Time (UTC+3:00)</td>
</tr>
<tr>
<td>EET</td>
<td>Eastern European Time (UTC+2:00)</td>
</tr>
<tr>
<td>JST</td>
<td>Japan Standard Time (UTC+9:00)</td>
</tr>
<tr>
<td>MDT</td>
<td>Mountain Daylight Time (UTC-6:00)</td>
</tr>
<tr>
<td>MST</td>
<td>(US) Mountain Standard Time (UTC-7:00)</td>
</tr>
<tr>
<td>NDT</td>
<td>Newfoundland Daylight Time (UTC-2:30)</td>
</tr>
<tr>
<td>NST</td>
<td>Newfoundland Standard Time (UTC-3:30)</td>
</tr>
<tr>
<td>NZDT</td>
<td>New Zealand Daylight Time (UTC+11:00)</td>
</tr>
<tr>
<td>NZST</td>
<td>New Zealand Standard Time (UTC+12:00)</td>
</tr>
<tr>
<td>NZT</td>
<td>New Zealand Time (UTC+12:00)</td>
</tr>
<tr>
<td>NFT</td>
<td>(Australian) Norfolk Island Time (UTC+11:30)</td>
</tr>
<tr>
<td>PDT</td>
<td>Pacific Daylight Time (UTC-7:00)</td>
</tr>
<tr>
<td>PST</td>
<td>(US) Pacific Standard Time (UTC-8:00)</td>
</tr>
<tr>
<td>SST</td>
<td>Singapore Standard Time (UTC+8:00)</td>
</tr>
<tr>
<td>WEDT</td>
<td>Western European Daylight Time (UTC+1:00)</td>
</tr>
<tr>
<td>WEST</td>
<td>Western European Summer Time (UTC+1:00)</td>
</tr>
<tr>
<td>WET</td>
<td>Western European Time (UTC)</td>
</tr>
<tr>
<td>WST</td>
<td>(Australian) Western Standard Time (UTC+8:00)</td>
</tr>
<tr>
<td>+/-hh:mm</td>
<td>is the displacement from UTC, for example, -8:00, 07:43</td>
</tr>
</tbody>
</table>

<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:mm:ss[ffffff]]. 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>])
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type</td>
</tr>
<tr>
<td>&lt;date format&gt;</td>
<td>The date format to use (only valid if &lt;value&gt; is a string)</td>
</tr>
</tbody>
</table>

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:mm: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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>A DATE or DATE WITH TIME ZONE value or the text representation of a date or date time</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;date&gt; if &lt;date&gt; is a string</td>
</tr>
</tbody>
</table>

**Examples**
- ToDate('2006-06-22 121314','yyy-mm-dd hhmiss')
  This example returns the date value 2006-06-22 12:13:14
- ToDate('2006-06-22 12:13:14 -5.00','hh:mm:ss stzh:tzm')
  This example returns 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**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type</td>
</tr>
<tr>
<td>&lt;scale&gt;</td>
<td>A number value that specifies the maximum number of digits that can follow the decimal point</td>
</tr>
</tbody>
</table>

**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>)**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type</td>
</tr>
</tbody>
</table>

### 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>)**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type except NUMBER</td>
</tr>
</tbody>
</table>

### 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 sdddddddddd hh:mm:ss[.ffffff]. 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])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>An INTERVAL DAY TO SECOND value or the text representation of the interval</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>A value of any numeric type</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;string&gt; if it is a string</td>
</tr>
</tbody>
</table>

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 syyyyyyyy-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))

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>An INTERVAL YEAR TO MONTH value or the text representation of the interval</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>An INTEGER value</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;string&gt;</td>
</tr>
</tbody>
</table>

**Examples**

- ToIntervalYM('1 11','syyyy 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any type</td>
</tr>
<tr>
<td>&lt;precision&gt;</td>
<td>A number value that specifies the maximum number of digits included in the number</td>
</tr>
</tbody>
</table>
### Symbol

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;scale&gt;</td>
<td>A number value that specifies the maximum number of digits that can follow the decimal point</td>
</tr>
</tbody>
</table>

### 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:mm: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>])`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;time&gt;</td>
<td>A TIME or TIME WITH TIME ZONE value or the text representation of the time</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>A value of any numeric data type</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;time&gt; if &lt;time&gt; is a string</td>
</tr>
</tbody>
</table>

### 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, 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 Data Manager is running.

Syntax

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

The time zone. Possible values are:

- **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)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEST</td>
<td>Central European Summer Time (UTC+2:00)</td>
</tr>
<tr>
<td>CET</td>
<td>Central European Time (UTC+1:00)</td>
</tr>
<tr>
<td>EDT</td>
<td>Eastern Daylight Time (UTC-4:00)</td>
</tr>
<tr>
<td>EST</td>
<td>(US) Eastern Standard Time (UTC-5:00)</td>
</tr>
<tr>
<td>EEDT</td>
<td>Eastern European Daylight Time (UTC+3:00)</td>
</tr>
<tr>
<td>EEST</td>
<td>Eastern European Summer Time (UTC+3:00)</td>
</tr>
<tr>
<td>EET</td>
<td>Eastern European Time (UTC+2:00)</td>
</tr>
<tr>
<td>JST</td>
<td>Japan Standard Time (UTC+9:00)</td>
</tr>
<tr>
<td>MDT</td>
<td>Mountain Daylight Time (UTC-6:00)</td>
</tr>
<tr>
<td>MST</td>
<td>(US) Mountain Standard Time (UTC-7:00)</td>
</tr>
<tr>
<td>NDT</td>
<td>Newfoundland Daylight Time (UTC-2:30)</td>
</tr>
<tr>
<td>NST</td>
<td>Newfoundland Standard Time (UTC-3:30)</td>
</tr>
<tr>
<td>NZDT</td>
<td>New Zealand Daylight Time (UTC+11:00)</td>
</tr>
<tr>
<td>NZST</td>
<td>New Zealand Standard Time (UTC+12:00)</td>
</tr>
<tr>
<td>NZT</td>
<td>New Zealand Time (UTC+12:00)</td>
</tr>
<tr>
<td>NFT</td>
<td>(Australian) Norfolk Island Time (UTC+11:30)</td>
</tr>
<tr>
<td>PDT</td>
<td>Pacific Daylight Time (UTC-7:00)</td>
</tr>
<tr>
<td>PST</td>
<td>(US) Pacific Standard Time (UTC-8:00)</td>
</tr>
<tr>
<td>SST</td>
<td>Singapore Standard Time (UTC+8:00)</td>
</tr>
<tr>
<td>WEDT</td>
<td>Western European Daylight Time (UTC+1:00)</td>
</tr>
<tr>
<td>WEST</td>
<td>Western European Summer Time (UTC+1:00)</td>
</tr>
<tr>
<td>WET</td>
<td>Western European Time (UTC)</td>
</tr>
<tr>
<td>WST</td>
<td>(Australian) Western Standard Time (UTC+8:00)</td>
</tr>
</tbody>
</table>

+/-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)
### Symbol Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A DATE, DATE WITH TIME ZONE, TIME, TIME WITH TIME ZONE, or CHAR value which is the text representation of a date or time</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;value&gt; if it is a string</td>
</tr>
</tbody>
</table>

### 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.

These are the control functions:

- `ArrayAddItem`
- `ArrayClear`
- `ArrayDeleteItem`  
- `ArrayItem`  
- `ArrayModifyItem`  
- `ArraySearch`  
- `ArraySize`  
- `ArraySort`  
- `Audit`  
- `AuditTrail`  
- `DBMS`  
- `DBName`  
- `Delay`  
- `Driver`
• Exit
• FileCheck
• FileClose
• FileFromParts
• FileFullPath
• FileList
• FileOpen
• FileRead
• FileWrite
• GetDirectory
• GetFileName
• LogMsg
• Lookup
• MessageCode
• MessageCount
• MessageSeverity
• MessageText
• NodeAuditId
• NodeStatus
• OpSys
• RowNum
• RowsInserted
• RowsUpdated
• SendAlert
• SendMail
• SQL
• SQLBind
• SQLClose
• SQLColumnCount
• SQLColumnName
In addition to these functions, 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" (p. 49).

**ArrayAddItem**

Adds an element to an array.

For more information, see "Data Types in Variables" (p. 112).

**Syntax**

ArrayAddItem(<array>, <value>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A variable of type ARRAY</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>The value to add which can be any data type</td>
</tr>
</tbody>
</table>

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A variable of type ARRAY</td>
</tr>
</tbody>
</table>
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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>The ordinal position, starting from one, of the element to delete</td>
</tr>
</tbody>
</table>

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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>The ordinal position, starting from one, of the required element</td>
</tr>
</tbody>
</table>

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> )

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
<tr>
<td>&lt;index&gt;</td>
<td>The ordinal position, starting from one, of the element to modify</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>The new value to use which can be any data type</td>
</tr>
</tbody>
</table>

**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" (p. 27).

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>The value to search for</td>
</tr>
<tr>
<td>&lt;sorted&gt;</td>
<td>Specify TRUE or FALSE to indicate whether the array has been sorted</td>
</tr>
</tbody>
</table>

**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.

- \texttt{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**

\texttt{ArraySize(<array>)}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
</tbody>
</table>

**Examples**

\texttt{$num\_elements := ArraySize($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.

Returns the number of items in the array.

**Syntax**

\texttt{ArraySort (<array>)}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;array&gt;</td>
<td>A value of type ARRAY</td>
</tr>
</tbody>
</table>

For more information, see "ArraySearch" (p. 26).

**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**

\texttt{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.

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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;audit_id&gt;</td>
<td>An integer that identifies a specific execution of a fact build or dimension build.</td>
</tr>
<tr>
<td>&lt;group&gt;</td>
<td>The type of audit information required. For a list of permitted values and descriptions, see &quot;Group Values&quot; (p. 29).</td>
</tr>
<tr>
<td>&lt;item&gt;</td>
<td>The item for which you require information. The value of &lt;group&gt; 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 &quot;Item Values&quot; (p. 29).</td>
</tr>
</tbody>
</table>
| <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

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACQUIRE</td>
<td>Information about source data acquisition. For example, the number of data rows that each data source contributes.</td>
</tr>
<tr>
<td>TRANSFORM</td>
<td>Information about the transformation of data. For example, the number of data rows that the transformation engine receives from each acquisition module.</td>
</tr>
<tr>
<td>DELIVER</td>
<td>Information about data delivery. For example, the number of data rows delivered.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>Information about internal IBM® Cognos® Data Manager structures. For example, the size of the hash table.</td>
</tr>
<tr>
<td>TIMING</td>
<td>Timing information. For example, the build start and end times, and the elapsed time for the build.</td>
</tr>
</tbody>
</table>

### Item Values

**ACQUIRE Group**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWLIMITS</td>
<td>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.</td>
</tr>
<tr>
<td>ROWS</td>
<td>A string that gives the name of a data source together with the three numbers, for example, '[Sales] 1000 2000 1000'.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The first number specifies the physical number of rows that the data source retrieved from the database.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>The third number specifies the number of output DataStream rows to which the data source contributes.</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>READ</td>
<td>The number of data rows that the DataStream makes available to the transformation engine.</td>
</tr>
<tr>
<td>ACCEPTED</td>
<td>The number of data rows that IBM® Cognos® Data Manager accepts.</td>
</tr>
<tr>
<td>REJECTED</td>
<td>The number of data rows that Data Manager rejects.</td>
</tr>
<tr>
<td>DIRECT</td>
<td>The number of candidate output data rows that come directly from the source data.</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>The number of candidate output data rows that come from consolidation of the source data.</td>
</tr>
<tr>
<td>TOTAL</td>
<td>The total number of candidate output data rows.</td>
</tr>
</tbody>
</table>

**DELIVER Group**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Specifies an identifier, a name, and the delivery module used for a delivery. For example, '[1] SalesFact (TABLE)'.</td>
</tr>
<tr>
<td>ROWS</td>
<td>The identifier and number of data rows delivered using a delivery. For example, '[1] 1250'.</td>
</tr>
</tbody>
</table>

**INTERNAL Group**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN_HASH_TABLE_SIZE</td>
<td>The minimum size of the hash table, in slots, to prevent resizing of this table during build execution.</td>
</tr>
<tr>
<td>PAGE_FAULTS</td>
<td>The number of page faults that occurred while executing the build.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>The amount of memory, in megabytes, required to execute the build.</td>
</tr>
</tbody>
</table>

**TIMING Group**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>A string value of the form, &quot;Starting Build '&lt;buildname&gt;'&quot;.</td>
</tr>
</tbody>
</table>
### Item Description

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELAPSED_TIME</td>
<td>A string value of the form, &quot;&lt;d&gt; days, &lt;h&gt; hours, &lt;m&gt; mins, &lt;s&gt; secs&quot;, that gives the total time taken to execute the build.</td>
</tr>
<tr>
<td>ELAPSED_SECS</td>
<td>The number of seconds taken to execute the build.</td>
</tr>
<tr>
<td>END-SUCCESS</td>
<td>A string value that indicates successful completion of the build. For example, 'Build completed successfully'.</td>
</tr>
<tr>
<td>END-FAILURE</td>
<td>A string value that indicates build failure. For example, 'Build failed'.</td>
</tr>
</tbody>
</table>

### Examples

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

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.

### 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®
- MSFTS
- 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alias&gt;</td>
<td>A database connection or alias</td>
</tr>
</tbody>
</table>

**Examples**

DBMS('GO_Sales')

This example returns SYBASE

**DBName**

Returns the database name for the alias.

**Syntax**

DBName(<alias>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alias&gt;</td>
<td>A database alias</td>
</tr>
</tbody>
</table>

**Examples**

DBName ('GO_Sales')

This example returns GOSales

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;seconds&gt;</td>
<td>A positive integer that represents the number of seconds to pause</td>
</tr>
</tbody>
</table>

**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alias&gt;</td>
<td>A database connection or alias</td>
</tr>
</tbody>
</table>

**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 Data Manager and is only for use by your scripts or other processes.

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;code&gt;</td>
<td>An integer</td>
</tr>
</tbody>
</table>

**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**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filename&gt;</td>
<td>The full path and file name of a file.</td>
</tr>
<tr>
<td>&lt;property&gt;</td>
<td>A string that specifies a property of the file to test. These are the valid properties</td>
</tr>
<tr>
<td></td>
<td>• 'EXISTS' specifies that IBM Cognos® Data Manager can find the file</td>
</tr>
<tr>
<td></td>
<td>• 'READ' specifies that Data Manager can read from the file</td>
</tr>
<tr>
<td></td>
<td>• 'WRITE' specifies that Data Manager can write to the file</td>
</tr>
<tr>
<td></td>
<td>• 'EXECUTE' specifies that the file is executable</td>
</tr>
</tbody>
</table>

**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, 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file_no&gt;</td>
<td>The number of the file to close</td>
</tr>
</tbody>
</table>

**Note:** Use the number returned when you open the file. For more information, see "FileOpen" (p. 37).

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;directory&gt;</td>
<td>The directory for the file</td>
</tr>
<tr>
<td>&lt;filename&gt;</td>
<td>The name of the file, without an extension</td>
</tr>
<tr>
<td>&lt;extension&gt;</td>
<td>The extension of the file</td>
</tr>
</tbody>
</table>

**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** function to create the file.

### FileFullPath

**Converts a relative file path to an absolute file path.**

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;full_path&gt;</td>
<td>A file path, which may be relative</td>
</tr>
</tbody>
</table>

**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]])
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filename&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;full&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;drill&gt;</td>
<td>This parameter specifies whether to search subdirectories. Set this to TRUE to search subdirectories and to FALSE to omit them.</td>
</tr>
</tbody>
</table>

**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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filename&gt;</td>
<td>A string value that specifies the files to open.</td>
</tr>
<tr>
<td>&lt;mode&gt;</td>
<td>The mode for which you want to open the file</td>
</tr>
<tr>
<td></td>
<td>● READ specifies open the file for reading</td>
</tr>
<tr>
<td></td>
<td>● WRITE specifies open the file for writing, overwriting the file if it exists</td>
</tr>
<tr>
<td></td>
<td>● APPEND specifies open an existing file for writing, appending to the file if it exists</td>
</tr>
</tbody>
</table>

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file_no&gt;</td>
<td>The number of the file from which to read</td>
</tr>
</tbody>
</table>

**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...>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file_no&gt;</td>
<td>The number of the file to which to write.</td>
</tr>
</tbody>
</table>
| <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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;full_path&gt;</td>
<td>The path from which to extract directory information</td>
</tr>
</tbody>
</table>

**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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;full_path&gt;</td>
<td>The file path from which to extract file name information</td>
</tr>
<tr>
<td>&lt;strip&gt;</td>
<td>Specify TRUE or FALSE to indicate whether or not to remove the file extension from the file name</td>
</tr>
</tbody>
</table>

**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...>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A text string which may contain variables</td>
</tr>
</tbody>
</table>

**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])
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alias&gt;</td>
<td>A database connection or alias.</td>
</tr>
<tr>
<td>&lt;statement&gt;</td>
<td>An SQL statement.</td>
</tr>
<tr>
<td>&lt;quote&gt;</td>
<td>Either TRUE or FALSE to indicate whether or not to enclose each returned value in quotation marks. The default value is FALSE.</td>
</tr>
<tr>
<td>&lt;max_values&gt;</td>
<td>Limits the number of items that are retrieved.</td>
</tr>
</tbody>
</table>

**Examples**

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

<table>
<thead>
<tr>
<th>ProdNo</th>
<th>Description</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0001</td>
<td>Camping Kettle</td>
<td>2.45</td>
</tr>
</tbody>
</table>

```
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>)`  

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;audit_id&gt;</code></td>
<td>The audit ID of the required fact build, dimension build, or JobStream</td>
</tr>
<tr>
<td><code>&lt;message_no&gt;</code></td>
<td>The number of the message for which the code is required</td>
</tr>
</tbody>
</table>

**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>)`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;audit_id&gt;</code></td>
<td>The audit ID of the execution process</td>
</tr>
</tbody>
</table>

**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

```plaintext
$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>)`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;audit_id&gt;</td>
<td>The audit ID of the execution process to which the message relates</td>
</tr>
<tr>
<td>&lt;message_no&gt;</td>
<td>Starting from 1, the number of the message for which the text is required</td>
</tr>
</tbody>
</table>

#### Examples

```plaintext
$msg_txt := MessageText($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>)`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;node_id&gt;</td>
<td>The identifier of the JobStream fact build or dimension build node.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>The node is processing</td>
</tr>
<tr>
<td>S</td>
<td>The node completed successfully</td>
</tr>
<tr>
<td>F</td>
<td>The node failed</td>
</tr>
</tbody>
</table>

Syntax

NodeStatus(<node_id>)

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.

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

**Syntax**

\[
\text{RowsInserted(</audit\_id>[, <table\_name>[, <dbalias>]])}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;audit_id&gt;</td>
<td>An integer that identifies a specific execution of a specific fact or dimension build</td>
</tr>
<tr>
<td>&lt;table_name&gt;</td>
<td>The name of a target data table</td>
</tr>
<tr>
<td>&lt;dbalias&gt;</td>
<td>The name of a database connection</td>
</tr>
</tbody>
</table>

**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**

\[
\text{RowsUpdated(</audit\_id>[, <table\_name>[, <dbalias>]])}
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;audit_id&gt;</td>
<td>An integer that identifies a specific execution of a specific fact or dimension build</td>
</tr>
<tr>
<td>&lt;table_name&gt;</td>
<td>The name of a target data table</td>
</tr>
<tr>
<td>&lt;dbalias&gt;</td>
<td>The name of a database connection</td>
</tr>
</tbody>
</table>

**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® BI.

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

Syntax
SendAlert(<item>, <value>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;item&gt;</td>
<td>The user-defined keyword to which the message relates</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>The text of the audit message</td>
</tr>
</tbody>
</table>

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>]])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;profile&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;password&gt;</td>
<td>The password that you use to access email.</td>
</tr>
<tr>
<td>&lt;subject&gt;</td>
<td>The subject text of the email.</td>
</tr>
</tbody>
</table>
### 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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;alias&gt;</td>
<td>A database connection or alias</td>
</tr>
<tr>
<td>&lt;statement&gt;</td>
<td>An SQL statement</td>
</tr>
</tbody>
</table>
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**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;command&gt;</td>
<td>An operating system command</td>
</tr>
</tbody>
</table>

### Examples

- **On Windows®**
  
  ```plaintext
  System('del d:\temp\temp.txt')
  System('copy d:\temp\file1.txt d:\temp\file2.txt')
  System('dir c:\temp\file2.txt')
  ```

- **On UNIX®**
  
  ```plaintext
  System('rm /tmp/temp.txt')
  System('cp /tmp/file1.txt /tmp/file2.txt')
  ```

### UUID

Returns a Universally Unique Identifier (UUID) string.
VariableInfo

Returns the data type and associated information about a variable.

For any variable, you can use the TypeInfo 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable&gt;</td>
<td>The variable for which you want type information</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>A string value that specifies the type of information you want</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DATATYPE'</td>
<td>The function returns a string value that names the data type of the &lt;variable&gt; 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'.</td>
</tr>
<tr>
<td>'PRECISION'</td>
<td>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.</td>
</tr>
<tr>
<td>'SCALE'</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

The functions are listed below, in the order that they are used:

- SQLPrepare
- SQLGetLastError
- SQLColumnCount
- SQLColumnName
- SQLColumnNo
- SQLBind
- SQLData
- SQLFetch
- SQLClose

**Examples**

```
$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 );
```
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.

### 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>] )`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;alias&gt;</code></td>
<td>A database connection or alias.</td>
</tr>
<tr>
<td><code>&lt;SQL&gt;</code></td>
<td>An SQL SELECT statement.</td>
</tr>
<tr>
<td><code>&lt;CognosSQL&gt;</code></td>
<td>The dialect of SQL to use, either native or Cognos® SQL.</td>
</tr>
<tr>
<td><code>&lt;comment&gt;</code></td>
<td>Optional comment about the SELECT statement. This comment is included in</td>
</tr>
</tbody>
</table>

**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.
Examples
$sqclid := 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" (p. 49). 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 sqclid variable. All subsequent operations on the cursor should use the integer value stored in the sqclid variable, until the cursor is closed.

**SQLGetLastError**

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

**Syntax**

SQLGetLastError()

**Examples**

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

In this extract, from the example script in "SQL Cursor Functions" (p. 49), 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> )

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CursorId&gt;</td>
<td>The cursor variable assigned to the SELECT statement</td>
</tr>
</tbody>
</table>

**Examples**

SQLColumnCount( $sqclid )

In this extract, from the example script in "SQL Cursor Functions" (p. 49), 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 "SQL-ColumnNo" (p. 52) function instead.

Syntax

\[
\text{SQLColumnName}( \text{<CursorId>}, \text{<ColumnNo>} )
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{&lt;CursorId&gt;}</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
<tr>
<td>\text{&lt;ColumnNo&gt;}</td>
<td>The column number for which you want to obtain the column name</td>
</tr>
</tbody>
</table>

Examples

\[
\begin{align*}
\text{$linetext} & := ''; \\
\text{$i} & := 0; \\
\text{while $i < $nColumns do} & \\
\text{begin} & \\
\text{$i} & := $i + 1; \\
\text{if $i > 1 then} & \text{$linetext} := \text{concat( $linetext, ',')} \\
\text{$linetext} & := \text{concat( $linetext, \text{SQLColumnName( $sqlid, $i )} )}; \\
\text{end}
\end{align*}
\]

In this extract, from the example script in "SQL Cursor Functions" (p. 49), 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 "SQLColumn-Name" (p. 51) function instead.

Syntax

\[
\text{SQLColumnNo}( \text{<CursorId>}, \text{<ColumnName>} )
\]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{&lt;CursorId&gt;}</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
<tr>
<td>\text{&lt;ColumnName&gt;}</td>
<td>The column name for which you want to obtain the column number</td>
</tr>
</tbody>
</table>

Examples

\[
\text{SQLColumnNo( $cursorid, 'Amount')}
\]

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**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CursorId&gt;</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
<tr>
<td>&lt;BindNo&gt;</td>
<td>The parameter number to bind</td>
</tr>
<tr>
<td>&lt;BindValue&gt;</td>
<td>The value to bind to a parameter</td>
</tr>
</tbody>
</table>

**Examples**

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

In this extract, from the example script in "SQL Cursor Functions" (p. 49), 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**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CursorId&gt;</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
</tbody>
</table>

**Examples**

```plaintext
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" (p. 49), each row of data is fetched.

**SQLData**

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

**Syntax**

SQLData( <CursorId>, <ColumnNo> )

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CursorId&gt;</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
<tr>
<td>&lt;ColumnNo&gt;</td>
<td>The column number for which to obtain the value</td>
</tr>
</tbody>
</table>

**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" (p. 49), 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> )

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CursorId&gt;</td>
<td>The cursor opened for the SELECT statement</td>
</tr>
</tbody>
</table>

**Examples**

SQLClose( $sqlid );

In this extract, from the example script in "SQL Cursor Functions" (p. 49), the cursor for the SELECT statement is closed.

**Logical Functions**

Logical functions return their results dependent upon some test.

These are the logical functions:

- **Choose**
- **If**
- **IfNull**

**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...])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;index&gt;</td>
<td>An integer that denotes which value to return from the list of outcomes</td>
</tr>
<tr>
<td>&lt;outcome1&gt;</td>
<td>The value returned if index=1</td>
</tr>
<tr>
<td>&lt;outcome2&gt;</td>
<td>The value returned if index=2</td>
</tr>
</tbody>
</table>
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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A logical expression</td>
</tr>
<tr>
<td>&lt;TRUE outcome&gt;</td>
<td>The expression returned if &lt;value&gt; evaluates to TRUE</td>
</tr>
<tr>
<td>&lt;FALSE outcome&gt;</td>
<td>The expression returned if &lt;value&gt; evaluates to FALSE</td>
</tr>
</tbody>
</table>

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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A logical expression</td>
</tr>
<tr>
<td>&lt;NULL outcome&gt;</td>
<td>The value returned if &lt;value&gt; is NULL</td>
</tr>
</tbody>
</table>

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.

These are the mathematical functions:
Abs

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

Syntax

Abs(<number>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>Any numeric value</td>
</tr>
</tbody>
</table>

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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numerical value</td>
</tr>
<tr>
<td>&lt;min&gt;</td>
<td>The minimum number into which Band places &lt;number&gt;</td>
</tr>
<tr>
<td>&lt;grain&gt;</td>
<td>The size of the bands into which Band divides the set</td>
</tr>
</tbody>
</table>

**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

**Examples**

- `Ceil(8.35)`
  This example returns 9

**Cos**

Returns the cosine of an angle.
Syntax
Cos(<number>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value that represents an angle expressed in radians</td>
</tr>
</tbody>
</table>

Examples
Cos(10)
This example returns -0.839072

Exp

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

Syntax
Exp(<number>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

Examples
Floor(8.75)
This example returns 8

Ln

Returns the natural logarithm (to base \(e\)) of the specified number.
### Syntax
\[ \text{Ln}(<\text{number}>) \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{number}&gt;)</td>
<td>A numeric value greater than 0</td>
</tr>
</tbody>
</table>

### Examples
\[ \text{Ln}(10) \]
This example returns 2.302585

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

### Syntax
\[ \text{Log}(<\text{number}>) \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{number}&gt;)</td>
<td>A numeric value greater than 0</td>
</tr>
</tbody>
</table>

### Examples
\[ \text{Log}(123) \]
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
\[ \text{Mod}(<\text{number}>, <\text{number}>) \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{number}&gt;)</td>
<td>An integer</td>
</tr>
<tr>
<td>(&lt;\text{number}&gt;)</td>
<td>An integer</td>
</tr>
</tbody>
</table>

### Examples
\[ \text{Mod}(10, 3) \]
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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

**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 \(<\text{min}> + n<\text{grain}>\) where \(<\text{min}>\) is the specified minimum number, \(n\) is some integer, and \(<\text{grain}>\) is a specified interval.

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

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;min&gt;</td>
<td>The minimum value of the generated random number</td>
</tr>
<tr>
<td>&lt;max&gt;</td>
<td>The maximum value of the generated random number</td>
</tr>
<tr>
<td>&lt;grain&gt;</td>
<td>The interval between members of the random number series</td>
</tr>
<tr>
<td>&lt;seed&gt;</td>
<td>An optional value with which to seed the random number generator</td>
</tr>
</tbody>
</table>

**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, Data Manager rounds off digits to the left of the decimal point.

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>The number to round</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>An integer; the number of decimal places to round</td>
</tr>
</tbody>
</table>

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

**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**

\[ \text{Sin(} \text{<number>} \text{)} \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>An angle expressed in radians</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{Sin(10)} \]

This example returns -0.544021

**Sqrt**

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

**Syntax**

\[ \text{Sqrt(} \text{<number>} \text{)} \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
</tbody>
</table>

**Examples**

\[ \text{Sqrt(12.4)} \]

This example returns 3.521363

**Tan**

Returns the tangent of the specified angle.

**Syntax**

\[ \text{Tan(} \text{<number>} \text{)} \]
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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>An integer value</td>
</tr>
</tbody>
</table>

#### Examples

- Trunc(22.653, 1)
  
  This example returns 22.6

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

### Member Functions

Note: Because member functions refer to the IBM® Cognos® Data Manager dimensional framework, they are valid only within the context of Data Manager builds. You cannot use these functions when acquiring data from an SQLTXT database.

These are the member functions

- IsAncestor
- Level
- Member
- TypeInfo
**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>')

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dimension&gt;</td>
<td>The name of a dimension element of the current fact build</td>
</tr>
<tr>
<td>&lt;member_id&gt;</td>
<td>The identifier of the possible ancestor of the current member</td>
</tr>
</tbody>
</table>

**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>')

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dimension&gt;</td>
<td>The name of a dimension within the current hierarchy</td>
</tr>
<tr>
<td>&lt;property&gt;</td>
<td>The name of the property that this function should return</td>
</tr>
<tr>
<td></td>
<td>•  NAME specifies the name of the level of the current member in the specified dimension</td>
</tr>
<tr>
<td></td>
<td>•  CAPTION specifies the business name of the level of the current member in the specified dimension</td>
</tr>
</tbody>
</table>
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>')

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dimension&gt;</td>
<td>The name of the dimension within the current hierarchy.</td>
</tr>
<tr>
<td>&lt;property&gt;</td>
<td>The name of the property to be returned. This can be one of the following properties</td>
</tr>
<tr>
<td></td>
<td>ID specifies the identifier of the member.</td>
</tr>
<tr>
<td></td>
<td>CAPTION specifies the identifier of the caption of the member.</td>
</tr>
<tr>
<td></td>
<td>PARENT specifies the identifier of the parent of the member.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

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>)
Description Symbol
The transformation model element for which you want type information
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.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DATATYPE'</td>
<td>The function returns a string value that names the data type of the &lt;element&gt; 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'.</td>
</tr>
<tr>
<td>'PRECISION'</td>
<td>For CHAR elements, the function returns an integer that gives the maximum number of characters that the element can contain.</td>
</tr>
<tr>
<td></td>
<td>For NUMBER elements, the function returns an integer that gives the maximum number of digits that the element can contain.</td>
</tr>
<tr>
<td></td>
<td>For other data types, the function returns zero.</td>
</tr>
<tr>
<td>'SCALE'</td>
<td>For NUMBER elements, the function returns an integer that gives the maximum number of digits that can follow the decimal point.</td>
</tr>
<tr>
<td></td>
<td>For other data types, the function returns zero.</td>
</tr>
</tbody>
</table>

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)
### Text Functions

These are the text functions:

- Char
- Checksum
- Collapse
- Concat
- ConcatSep
- CountStr
- ExtractStr
- I18NConvert
- I18NString
- InitCap
- InStr
- IsAlpha
- IsAlphaNumeric
- IsDigit
- IsFloat
- IsInteger
- IsLower
- IsNumeric
- IsUpper
- Left
- Length
- Lower
- LPad
- LTrim
Char

Returns the ASCII character with the specified code.

Syntax
Char(<number>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;number&gt;</td>
<td>The numeric code of an ASCII character</td>
</tr>
</tbody>
</table>

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...>])
### 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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

#### 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

```plaintext
Concat(<value>, <value...>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type</td>
</tr>
</tbody>
</table>

#### 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...>])
```

**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>)
```

**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>])`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value that contains a separated list of items</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>The ordinal position of the item that you want to extract from &lt;string&gt;</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>The character that separates the items within &lt;string&gt;</td>
</tr>
</tbody>
</table>

**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>)`

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value that contains a separated list of items</td>
</tr>
<tr>
<td>&lt;encoding&gt;</td>
<td>The encoding to be used, for example:</td>
</tr>
<tr>
<td></td>
<td>UTF-8 (the default)</td>
</tr>
<tr>
<td></td>
<td>UTF-16</td>
</tr>
<tr>
<td></td>
<td>US-ASCII</td>
</tr>
<tr>
<td></td>
<td>Shift-JIS</td>
</tr>
<tr>
<td></td>
<td>ISO-8859-1</td>
</tr>
</tbody>
</table>
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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value that contains a separated list of items</td>
</tr>
<tr>
<td>&lt;encoding&gt;</td>
<td>The encoding to be used to decode the string, for example:</td>
</tr>
<tr>
<td></td>
<td>• UTF-8 (the default)</td>
</tr>
<tr>
<td></td>
<td>• UTF-16</td>
</tr>
<tr>
<td></td>
<td>• US-ASCII</td>
</tr>
<tr>
<td></td>
<td>• Shift-JIS</td>
</tr>
<tr>
<td></td>
<td>• ISO-8859-1</td>
</tr>
</tbody>
</table>

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>)
```
### 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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string1&gt;</td>
<td>The string to be searched.</td>
</tr>
<tr>
<td>&lt;string2&gt;</td>
<td>The string for which <code>InStr</code> searches.</td>
</tr>
<tr>
<td>&lt;number1&gt;</td>
<td>The starting position of the search.</td>
</tr>
<tr>
<td>&lt;number2&gt;</td>
<td>An integer that determines for which occurrence of the second string the function searches.</td>
</tr>
</tbody>
</table>

**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>
**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

**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>)
```
### 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

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

#### 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

**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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

### 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A string value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>The number of characters to return</td>
</tr>
</tbody>
</table>

### 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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A value of any data type</td>
</tr>
</tbody>
</table>
Examples
Length('text')
This example returns 4

Lower
Casts a string to lowercase.

Syntax
Lower(<string>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>The length (in number of characters) of the returned string</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>The character (or character sequence) with which to left-pad the first string</td>
</tr>
</tbody>
</table>

Examples
- LPad('x',5,'*')
  This example returns '*****x'
- Lpad('text',11,'the')
  This example returns 'thethetext'
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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>A string value that gives the set of characters to remove from the first string</td>
</tr>
</tbody>
</table>

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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value.</td>
</tr>
<tr>
<td>&lt;search&gt;</td>
<td>The string value for which to search.</td>
</tr>
</tbody>
</table>
### Symbol

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;replace&gt;</code></td>
<td>The string value that replaces <code>&lt;search&gt;</code>. By default this is an empty string.</td>
</tr>
</tbody>
</table>

### 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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;string&gt;</code></td>
<td>A string value</td>
</tr>
<tr>
<td><code>&lt;number&gt;</code></td>
<td>The number of characters to return</td>
</tr>
</tbody>
</table>

**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 'textthethe'
- `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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>

**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>])

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
<tr>
<td>&lt;from&gt;</td>
<td>An integer that gives the position within &lt;string&gt; at which SubStr begins the returned string</td>
</tr>
<tr>
<td>&lt;count&gt;</td>
<td>An integer that gives the required length of the returned string</td>
</tr>
</tbody>
</table>

**Examples**

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

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

Syntax

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
<tr>
<td>&lt;old&gt;</td>
<td>A string value that contains the characters to be translated</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>A string value that provides the characters to replace the matching characters on &lt;old&gt;</td>
</tr>
</tbody>
</table>

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>])
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A string value</td>
</tr>
</tbody>
</table>
### Trim

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>)

**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.

**Notes**

- You cannot pass dates as text representations. To pass a text representation of a date to a date function, use the ToDate 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.

These are the date functions:

- AddDaysToInterval
- AddMonthsToDate
- AddMonthsToInterval
- AddSecondsToInterval
- AddToDate
- AddYearsToDate
- AddYearsToInterval
- DaysBetween
- FirstOfMonth
- IsLeapYear
- IsLeapYearDay
- IsValidDate
- IsValidIntervalDS
- IsValidIntervalYM
- IsValidTime
- LastOfMonth
- MonthsBetween
- SecondsBetween
- SysDate

**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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;interval&gt;</td>
<td>A day to second interval value</td>
</tr>
</tbody>
</table>
AddDaysToInterval

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

A date or date with time zone value

Syntax

AddMonthsToDate(<date>, <number>)

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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;interval&gt;</td>
<td>A year to month interval value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>An integer that gives the number of months to add to &lt;interval&gt;</td>
</tr>
</tbody>
</table>

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>)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;interval&gt;</td>
<td>A day to second interval value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>A numeric value that gives the number of seconds to add to &lt;interval&gt;</td>
</tr>
</tbody>
</table>

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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>A date or date with time zone value</td>
</tr>
<tr>
<td>&lt;number</td>
<td>interval&gt;</td>
</tr>
</tbody>
</table>

#### 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( '-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>)
```

---

90 IBM Cognos Data Manager
## AddYearsToDate

Returns a date resulting from adding the specified number of years to a date.

### Syntax

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>A date or date with time zone value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>An integer that gives the number of years to add to &lt;date&gt;</td>
</tr>
</tbody>
</table>

### Examples

- `AddYearsToDate(ToDate('2002-07-09','yyyy-mm-dd'),'2005-07-29','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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;interval&gt;</td>
<td>A year to month interval value</td>
</tr>
<tr>
<td>&lt;number&gt;</td>
<td>An integer that gives the number of years to add to &lt;interval&gt;</td>
</tr>
</tbody>
</table>

### 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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date</td>
<td>interval1&gt;</td>
</tr>
<tr>
<td>&lt;date</td>
<td>interval2&gt;</td>
</tr>
</tbody>
</table>

**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>)
```
### 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

\[ \text{IsLeapYear}\left(\text{<date>}\right) \]

#### Examples

- \[ \text{IsLeapYear}(\text{ToDate('2004-07-12','yyyy-mm-dd'))} \]
  
  This example returns TRUE

- \[ \text{IsLeapYear}(\text{ToDate('2005-07-12','yyyy-mm-dd'))} \]
  
  This example returns FALSE

---

### IsLeapYearDay

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

#### Syntax

\[ \text{IsLeapYearDay}\left(\text{<date>}\right) \]

#### Examples

- \[ \text{IsLeapYearDay}(\text{ToDate('2004-02-29','yyyy-mm-dd'))} \]
  
  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 sdddddddhh:mm:ss,.ffffff. 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>])
### IsValidIntervalDS

A function that checks if a given string represents a valid interval. The input value can be of type CHAR, INTERVAL YEAR TO MONTH, or INTEGER.

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A text representation of the interval</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;string&gt;</td>
</tr>
</tbody>
</table>

**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

A function that checks if a given string represents a valid year to month interval. The input value can be of type CHAR, INTERVAL YEAR TO MONTH, or INTEGER.

**Syntax**

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;string&gt;</td>
<td>A text representation of the interval</td>
</tr>
</tbody>
</table>
IsValidIntervalYM

Returns TRUE or FALSE to indicate whether a value represents a valid interval. 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:mm:ss[,ffffffff]. 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

IsValidIntervalYM returns TRUE for values of type TIME.

Syntax

IsValidIntervalYM(<value>[, <format>]))

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;value&gt;</td>
<td>A text representation of the time or TIME, TIME WITH TIME ZONE, or any numeric data type</td>
</tr>
<tr>
<td>&lt;format&gt;</td>
<td>The format of &lt;value&gt; if &lt;value&gt; is a string</td>
</tr>
</tbody>
</table>
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>)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;date&gt;</td>
<td>A date or date with time zone value</td>
</tr>
</tbody>
</table>

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>)
```
## MonthsBetween

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

### Syntax

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

### 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>)
```

### Description

- **<datetimelinterval1>**
  - A date, time, date with time zone, time with time zone, or day to second interval value
- **<datetimelinterval2>**
  - 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 <datetimelinterval1>)
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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;format&gt;</td>
<td>A string value that specifies a date format</td>
</tr>
</tbody>
</table>

**Examples**

```java
SysDate('dd-mmm-yyyy hh:mm: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

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

It supports 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. For example

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, Data Manager compares the left most character in each string. If these characters are equal, Data Manager compares the next character in each string. This continues until Data Manager finds a difference, or until there are no more characters to check, in one or both strings. If 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
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 operators
- mathematical operators

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equals. This binary operator returns TRUE if the operands to its left and right are equal. Otherwise, it returns FALSE.</td>
</tr>
<tr>
<td></td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>• 1=2 returns FALSE</td>
</tr>
<tr>
<td></td>
<td>• 3=3 returns TRUE</td>
</tr>
<tr>
<td>!=</td>
<td>Not equals. This binary operator returns TRUE if the operands to its left and right are not equal. Otherwise it returns FALSE.</td>
</tr>
<tr>
<td></td>
<td>Examples</td>
</tr>
<tr>
<td></td>
<td>• 1!=2 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 3!=3 returns FALSE</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>This binary operator is equivalent to !=.</td>
</tr>
</tbody>
</table>
### Operator Description

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>• 2&gt;=1 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 3&gt;=3 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 2&gt;=3 returns FALSE</td>
</tr>
<tr>
<td>&gt;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>• 2&gt;1 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 3&gt;3 returns FALSE</td>
</tr>
<tr>
<td>&lt;=</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>• 1&lt;=2 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 2&lt;=2 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 3&lt;=2 returns FALSE</td>
</tr>
<tr>
<td>&lt;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>• 1&lt;2 returns TRUE</td>
</tr>
<tr>
<td></td>
<td>• 2&lt;2 returns FALSE</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>IS</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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 |
| LIKE     | Like. This binary operator takes a string value as its left operand and a wildcarded 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. |
### Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOT</strong></td>
<td>Logical not. This unary operator takes a Boolean value as its operand and returns the inverse of the value.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>- NOT TRUE returns FALSE</td>
</tr>
<tr>
<td></td>
<td>- NOT (1 &gt; 3) returns TRUE</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>Logical and. This binary operator takes two Boolean operands. It returns TRUE if both operands are TRUE. Otherwise, it returns FALSE.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>- TRUE AND TRUE returns TRUE</td>
</tr>
<tr>
<td></td>
<td>- 1 &lt; 2 AND 'fred' LIKE 'f%' returns TRUE</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>Logical inclusive or. This binary operator takes two Boolean operands. It returns TRUE if either or both operands are TRUE. Otherwise, it returns FALSE.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>- TRUE OR TRUE returns TRUE</td>
</tr>
<tr>
<td></td>
<td>- (1 &lt; 2) OR (3 = 4) returns TRUE</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>- (2 + 4) / 3 = 6 / 3 = 2</td>
</tr>
<tr>
<td>&amp;</td>
<td>This operator is used when adding a list of values to an array, with each value separated by the ampersand (&amp;).</td>
</tr>
<tr>
<td></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td></td>
<td>- $ArrayVar := 'some text' &amp; 123.45</td>
</tr>
</tbody>
</table>
### Operator Description

- **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, Data Manager evaluates the expression from left to right.

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

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Brackets</td>
</tr>
<tr>
<td>7</td>
<td>Unary minus, that is, negation</td>
</tr>
</tbody>
</table>
### 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, 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.
  
  \[10 / 2 * 5\]

  Data Manager evaluates the expression from left to right, giving \(10 / 2 * 5 = 5 * 5 = 25\).

- In this example, `*` has the highest priority.
  
  \[5 + 2 * 5 - 2\]

  Data Manager first evaluates the multiplication, giving \(5 + 10 - 2\). Both remaining operators have priority 5. Therefore, 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.
  
  \[(5 + 2) * (5 - 2)\]

  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**)
- multi-way branch control (**CASE statements**)

---

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td><code>/</code></td>
</tr>
<tr>
<td>5</td>
<td><code>+ -</code></td>
</tr>
<tr>
<td>4</td>
<td>Logical comparisons, that is, <code>!=</code>, <code>&lt;&gt;</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>=</code>, <code>&lt;</code>, <code>IS</code>, <code>IN</code>, <code>LIKE</code>, <code>BETWEEN</code></td>
</tr>
<tr>
<td>3</td>
<td><code>NOT</code></td>
</tr>
<tr>
<td>2</td>
<td><code>AND</code></td>
</tr>
<tr>
<td>1</td>
<td><code>OR</code></td>
</tr>
</tbody>
</table>
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

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, 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 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

```
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\n  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. These are

- property variables, which are intrinsic to Data Manager and affect the operation of Data Manager commands
- user-defined variables, which store values for use in your scripts

You can use variables

- 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 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

The following data types are available for variables.

- ARRAY
- BOOLEAN
- BINARY
- CHAR(<n>)
- DATE
- DATE WITH TIMEZONE
- FLOAT
- INTEGER
- INTERVAL DAY TO SECOND
- INTERVAL YEAR TO MONTH
- NUMBER(<p>,<s>)
- TIME
- TIME WITH TIMEZONE

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
  
  $$\text{BinaryVar2} := \text{BinaryVar1}$$

The `ToHex` 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
  
  $$\text{TextVar} := \text{'variable'}$$

- 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
  
  $$\text{TextVar} := \text{'variable''s contents'}$$

- The value of another variable of type CHAR. For example
  
  $$\text{TextVar2} := \text{TextVar1}$$

- The result of using the `ToChar` function to change a value to a string representation of that value. The following example assigns '123' to TextVar if IntVar = 123
  
  $$\text{TextVar} := \text{ToChar($IntVar$)}$$

**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
  
  $$\text{DateVar2} := \text{DateVar1}$$

- The result of using the `ToDate` 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
  
  $$\text{DateVar} := \text{ToDate('12-15-2001', 'mm-dd-yyyy')}$$
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 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 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 function to cast a value to an integer. In the following example, 123 is assigned to IntVar if TextVar = '123'
$\text{IntVar} := \text{ToInteger($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
  $\text{IntDSVar2} := \text{IntDSVar1}$

- The result of using the \text{ToIntervalDS} 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.
  $\text{IntDSVar} := \text{ToIntervalDS('01 12:13:14', 'dd hh:mm:ss')}$

**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
  $\text{IntYMVar2} := \text{IntYMVar1}$

- The result of using the \text{ToIntervalYM} 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.
  $\text{IntYMVar} := \text{ToIntervalYM('010-10', 'yyyy-mm')}$

**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
  $\text{NumVar} := 123.45$

- The value of another variable of type NUMBER. For example
  $\text{NumVar2} := \text{NumVar1}$

- The result of a mathematical expression. For example
  $\text{NumVar} := (123.4*2.3)/5$

- The result of using the \text{ToDouble} or \text{ToInteger} 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 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, 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 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 Data Manager should perform substitution on a variable, enclose the variable in braces. For example, use [$Example] to specify that 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'';'

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 ($$3); **ELSE** RETURN ($$4);

If variables S1, S2, S3, and S4 are NULL when the script is parsed, substitution on this script produces the following:

```plaintext
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.

```plaintext
$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:

```plaintext
$RESULT := Lookup( 'GO_Sales', {$QrySpec} );
```

During substitution, it is replaced with the following statement:

```plaintext
$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:

```plaintext
$RESULT := Lookup( 'GO_Sales', '{$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:

```plaintext
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>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;statement_list&gt;</td>
<td>A sequence of statements in the IBM® Cognos® Data Manager scripting language.</td>
</tr>
<tr>
<td>&lt;statement&gt;</td>
<td>A single statement.</td>
</tr>
<tr>
<td>&lt;statement_block&gt;</td>
<td>A statement list that is enclosed in the BEGIN and END keywords. For example</td>
</tr>
<tr>
<td></td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>LogMsg('Procedure Successful');</td>
</tr>
<tr>
<td></td>
<td>$EXAMPLE := TRUE;</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
<tr>
<td></td>
<td>If the statement list contains only one statement, then the use of the BEGIN and END keywords is optional.</td>
</tr>
<tr>
<td>&lt;if_statement&gt;</td>
<td>A construct that conditionally executes a block of statements depending on the value of a Boolean expression. If the expression evaluates to</td>
</tr>
<tr>
<td></td>
<td>TRUE, then Data Manager executes the statement block that follows the THEN keyword, otherwise, it executes the statement block (if any) that</td>
</tr>
<tr>
<td></td>
<td>follows the ELSE keyword.</td>
</tr>
</tbody>
</table>
A construct that conditionally executes one of several statement blocks depending on the value of an expression. 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 Data Manager executes the corresponding statement block. If the result of evaluating the expression matches none of the literal values, then 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
```

A statement that assigns to a variable the result of executing an expression.

**Note:** This uses a Pascal-like assignment operator (:=).

A preconditioned control structure that executes a statement block while a logical expression evaluates to TRUE.

A statement that ends execution of the script and returns the value of the specified expression to the calling process.

A call to a built-in function or user-defined function.

```
<if_statement> ::= 
  IF <logical_expression> THEN <statement_block>
  ELSE <statement_block>
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
</table>
| `<case_statement>`  | A construct that conditionally executes one of several statement blocks depending on the value of an expression. 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 Data Manager executes the corresponding statement block. If the result of evaluating the expression matches none of the literal values, then 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 (:=).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;while_statement&gt;</code></td>
<td>A preconditioned control structure that executes a statement block while a logical expression evaluates to TRUE.</td>
</tr>
<tr>
<td><code>&lt;return_statement&gt;</code></td>
<td>A statement that ends execution of the script and returns the value of the specified expression to the calling process.</td>
</tr>
<tr>
<td><code>&lt;function_call&gt;</code></td>
<td>A call to a built-in function or user-defined function.</td>
</tr>
</tbody>
</table>
### THEN
This keyword introduces the statement block that Data Manager executes if the Boolean expression evaluates to TRUE.

### ELSE
If present, this keyword introduces the statement block that Data Manager executes if the Boolean expression evaluates to FALSE.

---

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>THEN</td>
<td>A block of statements.</td>
</tr>
<tr>
<td>ELSE</td>
<td>A block of statements.</td>
</tr>
</tbody>
</table>

---

### Logical Expression

- **Syntax:**

  \[
  \text{logical_expression} ::= \\
  \quad [\text{expression}] \text{logical_operator} \text{expression} \mid \\
  \quad (\text{logical_expression})
  \]

- **Symbols:**

  - **logical_operator**
    - AND
    - OR
    - NOT
    - !
    - =
    - !=
    - <
    - >
    - <=
    - >=

### Expression

- **Syntax:**

  \[
  \text{expression} ::= \\
  \quad \text{value} | \text{logical_expression} | \text{arith_expression} | \\
  \quad \text{function_call} | (\text{expression})
  \]

- **Symbols:**

  - **value**
    - A literal value or the value that a variable contains.
  - **variable**
    - A variable that stores a discrete value.
  - **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.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;literal&gt;</td>
<td>A discrete, literal value.</td>
</tr>
</tbody>
</table>

<arith_expression> ::= [[<expression>] <arith_operator> <expression>]| (<arith_expression>)

<arith_operator> ::= +|-|*|/

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;arith_expression&gt;</td>
<td>An arithmetic expression. That is, an expression that results in a numeric value.</td>
</tr>
<tr>
<td>&lt;expression&gt;</td>
<td>A literal value or an expression that evaluates to a value.</td>
</tr>
<tr>
<td>&lt;arith_operator&gt;</td>
<td>A mathematical operator. For information on the available logical operators, see &quot;Logical Operators&quot; (p. 103). For information on how Data Manager compares values, see &quot;Comparison of Values in Scripts&quot; (p. 102).</td>
</tr>
</tbody>
</table>

<function_call> ::= <function_name>({[[<value>]}|, <value>})

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;function_call&gt;</td>
<td>A call to a built-in function or user-defined function.</td>
</tr>
<tr>
<td>&lt;function_name&gt;</td>
<td>The name of the function that the line of syntax calls.</td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>An actual parameter of the function.</td>
</tr>
</tbody>
</table>

<case_statement> ::= CASE <expression> OF BEGIN <case_list> END

<case_list> ::= <literal>: <statement_block> {<literal>: <statement_block>} [DEFAULT: <statement_block>]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;case_statement&gt;</td>
<td>A construct that conditionally executes one of several statement blocks depending on the value of an expression.</td>
</tr>
</tbody>
</table>
### Symbol | Description
--- | ---
<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. 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 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 Data Manager tests to determine whether to execute the statement block.
<statement_block> | The block of statements that 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.
<expression> | An expression that evaluates to the value that Data Manager returns to the calling process.

<assignment> ::=  
<variable> := <expression>
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

```plaintext
$TestVar := 'This is a test line for the log';
LogMsg(Concat('$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 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 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 Data Manager Designer, on the command line, or using an environment variable.

**Steps using Data Manager Designer**

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.

**Step Using the Command Line**
- 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`

**Step Using an Environment Variable**
- Set the environment variable TRACE_VALUES to include USER.
  
  Note: In Windows®, you must set the TRACE_VALUES environment variable before starting 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" (p. 125)
- "Hierarchical Debug Categories" (p. 125)
- "Multiple Debug Categories" (p. 126)

**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:

```plaintext
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:

```plaintext
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 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. In such cases, 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.
Index

A
Abs function, 57
AddDaysToInterval function, 87
addition operator, 106
AddMonthsToDate function, 88
AddMonthsToInterval function, 89
AddSecondsToInterval function, 89
AddToDate function, 90
AddYearsToDate function, 90
AddYearsToInterval function, 91
AND operator, 103
arithmetic expression, syntax of, 122
ArrayAddItem function, 24
ArrayClear function, 24
ArrayDeleteItem function, 25
ArrayItem function, 25
ArrayModifyItem function, 26
ArraySearch function, 26
ArraySize function, 27
ArraySort function, 27
assignment operator, 101
assignment statement, syntax of, 123
Audit function, 27
AuditTrail function, 28

B
Band function, 57
BETWEEN operator, 103
brackets operator, 106
branching in scripts, 108

C
case statement, syntax of, 122
CASE statements, 109
Ceil function, 58
Char function, 69
Checksum function, 69
Choose function, 55
Collapse function, 70
comments in scripts, 101
comparison values in scripts, 102
characters, 102
dates, 102
nulls, 102
numeric, 102
string, 102
times, 102
Concat function, 70
ConcatSep function, 71
control functions, 22
controls
   branching, 108
Cos function, 58
CountStr function, 71

D
data types
   in scripts, 112
date functions, 86
daysBetween function, 92
DBMS function, 31
DBName, 32
debugging scripts, 124
Delay function, 32
division operator, 106
Driver function, 33

E
equals operator, 103
Exit function, 33
Exp function, 59
ExtractStr function, 71

F
FileCheck function, 34
FileClose function, 35
FileFromParts function, 35
FileFullPath function, 36
FileList function, 36
FileOpen function, 37
FileRead function, 37

Licensed Materials – Property of IBM
© Copyright IBM Corp. 2005, 2010.
**Index**

FileWrite function, 38
FirstOfMonth function, 92
Floor function, 59
function call, syntax of, 122
functions, 9
   Abs, 57
   AddDaysToInterval, 87
   AddMonthsToDate, 88
   AddMonthsToInterval, 89
   AddSecondsToInterval, 89
   AddToDate, 90
   AddYearsToDate, 90
   AddYearsToInterval, 91
ArrayAddItem, 24
ArrayClear, 24
ArrayDeleteItem, 25
ArrayItem, 25
ArrayModifyItem, 26
ArraySearch, 26
ArraySize, 27
ArraySort, 27
Audit, 27
AuditTrail, 28
Band, 57
Ceil, 58
Char, 69
Checksum, 69
Choose, 55
Collapse, 70
Concat, 70
ConcatSep, 71
control, 22
Cos, 58
CountStr, 71
date, 86
DaysBetween, 92
DBMS, 31
DBName, 32
Delay, 32
Driver, 33
Exit, 33
Exp, 59
ExtractStr, 71
FileCheck, 34
FileClose, 35
FileFromParts, 35
FileFullPath, 36
FileList, 36
FileOpen, 37
FileRead, 37
FileWrite, 38
FirstOfMonth, 92
Floor, 59
GetDirectory, 38
GetFileName, 39
I18NConvert, 72
I18NString, 73
If, 56
IfNull, 56
Initcap, 73
InStr, 74
IsAlpha, 75
IsAlphaNumeric, 75
IsAncestor, 65
IsDigit, 76
IsFloat, 76
IsInteger, 77
IsLeapYear, 93
IsLeapYearDay, 93
IsLower, 77
IsNumeric, 78
IsUpper, 78
IsValidDate, 94
IsValidIntervalDS, 94
IsValidIntervalYM, 95
IsValidTime, 96
LastOfMonth, 97
Left, 79
Length, 79
Level, 65
Ln, 59
Log, 60
logical, 55
LogMsg, 39
Lookup, 40
Lower, 80
LPad, 80
LTrim, 81
mathematical, 56
member, 64
Member, 66
MessageCode, 40
Index

MessageCount, 41
MessageSeverity, 41
MessageText, 42
Mod, 60
MonthsBetween, 97
NodeAuditID, 42
NodeStatus, 43
OpSys, 43
Power, 61
Rand, 61
Replace, 81
Right, 82
Round, 62
RowNum, 43
RowsInserted, 43
RowsUpdated, 44
RPad, 82
RTrim, 83
SecondsBetween, 98
SendAlert, 45
SendMail, 45
SetTimeZone, 10
Sign, 62
Sin, 63
Soundex, 83
SQL, 46
SQLBind, 53
SQLClose, 55
SQLColumnCount, 51
SQLColumnName, 51
SQLColumnNo, 52
SQL cursor, 49
SQLData, 54
SQLFetch, 53
SQLGetLastError, 51
SQLPrepare, 50
Sqrt, 63
SubStr, 84
SysDate, 99
System, 47
Tan, 63
text, 68
ToChar, 12
ToDate, 12
ToDateDouble, 13
ToHex, 14
ToInteger, 15
ToIntevalDS, 16
ToIntervalYM, 17
ToNumber, 17
ToTime, 18
ToTimeZone, 19
Translate, 85
Trim, 85
Trunc, 64
type conversion, 9
TypeInfo, 66
Upper, 86
UUID, 47
VariableInfo, 48

G
GetDirectory function, 38
GetFileName function, 39
greater than operator, 103

I
I18NConvert function, 72
I18NString function, 73
If function, 56
IfNull function, 56
IF statements, 109
Initcap function, 73
IN operator, 103
InStr function, 74
IsAlpha function, 75
IsAlphaNumeric function, 75
IsAncestor function, 65
IsDigit function, 76
IsFloat function, 76
IsInteger function, 77
IsLeapYearDay function, 93
IsLeapYear function, 93
IsLower function, 77
IsNumeric function, 78
IS operator, 103
IsUpper function, 78
IsValidDate function, 94
IsValidIntervalDS function, 94
IsValidIntervalYM function, 95
IsValidTime function, 96
Index

L
language, scripting, 101
LastOfMonth function, 97
Left function, 79
Length function, 79
less than operator, 103
less than or equals operator, 103
Level function, 65
LIKE operator, 103
Ln function, 59
Log function, 60
logical expression, syntax of, 121
logical functions, 55
logical operators, 103
LogMsg function, 39
Lookup function, 40
loops in scripts, 110
Lower function, 80
LPad function, 80
LTrim function, 81

M
mathematical functions, 56
mathematical operators, 106
Member function, 66
member functions, 64
MessageCode function, 40
MessageCount function, 41
MessageSeverity function, 41
MessageText function, 42
Mod function, 60
MonthsBetween function, 97
multiplication operator, 106

N
nested scripts, 111
NodeAuditID function, 42
NodeStatus function, 43
not equals operator, 103
NOT operator, 103

O
operators, 103
addition, 106
AND, 103
assignment, 101
BETWEEN, 103
brackets, 106
division, 106
equals, 103
greater than, 103
greater than or equals, 103
IN, 103
IS, 103
less than, 103
less than or equals, 103
LIKE, 103
logical, 103
mathematical, 106
multiplication, 106
NOT, 103
not equals, 103
OR, 103
order of precedence, 107
subtraction, 106
unary minus, 106
OpSys function, 43
order of precedence for operators, 107
OR operator, 103

P
Power function, 61

Q
quotation marks in scripts, 118

R
Rand function, 61
Replace function, 81
returned values in scripts, 102
return statement, syntax of, 123
Right function, 82
Round function, 62
RowNum function, 43
RowsInserted function, 43
RowsUpdated function, 44
RPad function, 82
RTrim function, 83

S
scripting
assignment operators, 101
Index

branch controls, 108
CASE statements, 109
character values, 102
comments, 101
comparison values, 102
date values, 102
debugging, 124
IF statements, 109
loops, 110
nesting, 111
null values, 102
numeric values, 102
operators, 103
quotation marks, 118
returned values, 102
string values, 102
substitution variables, 117
time values, 102
variable data types, 112
variables, 111
scripting language, 101
  arithmetic expression, 122
  assignment statement, 123
  case statement, 122
  expression, 121
  function call, 122
  if statement, 120
  logical expression, 121
  return statement, 123
  statement, 119
  while statement, 123
SecondsBetween function, 98
SendAlert function, 45
SendMail function, 45
SetTimeZone function, 10
Sign function, 62
Sin function, 63
Soundex function, 83
SQLBind function, 53
SQLClose function, 55
SQLColumnCount function, 51
SQLColumnName function, 51
SQLColumnNo function, 52
SQL Cursor functions, 49
SQLData function, 54
SQLFetch function, 53
SQL function, 46
SQLGetLastError function, 51
SQLPrepare function, 50
Sqrt function, 63
statement, syntax of, 119
substitution variables, 117
SubStr function, 84
subtraction operator, 106
syntax
  arithmetic expression, 122
  assignment statement, 123
  case statement, 122
  expression, 121
  function call, 122
  if statement, 120
  logical expression, 121
  return statement, 123
  scripting language, 119
  while statement, 123
SysDate function, 99
System function, 47

T
  Tan function, 63
text functions, 68
ToChar function, 12
ToDate function, 12
ToDate function, 12
ToDouble function, 13
ToHex function, 14
ToInteger function, 15
ToIntervalDS function, 16
ToIntervalYM function, 17
ToNumber function, 17
ToTime function, 18
ToTimeZone function, 19
Translate function, 85
Trim function, 85
Trunc function, 64
type conversion functions, 9
TypeInfo function, 66

U
  unary minus operator, 106
Upper function, 86
UUID function, 47
Index

V
VariableInfo function, 48
variables
data types, 112
in scripts, 111
substitution, 117

W
while statement, syntax of, 123