expeval: Expression Evaluation User Guide

CML00091-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com
Copyright © 2014 – 2024 by Code Magus Limited
All rights reserved

July 30, 2024

Contents

1 Overview **2**
1.1 Introduction 2

2 Expression Evaluation Syntax **2**
2.1 Expression Overview 2
2.2 Expression Grammar 3
2.2.1 Lexical Elements 3
2.2.2 Syntactical Elements 4
2.3 Built-in Functions 9
2.3.1 SysStrLen, strlen, length 9
2.3.2 SysSubStr, substr 10
2.3.3 SysString, string 10
2.3.4 SysNumber, number 10
2.3.5 SysStrCat, strcat 11
2.3.6 SysStrStr, strstr 11
2.3.7 SysStrSpn, strspn 12
2.3.8 SysStrCspn, strcspn 12
2.3.9 SysStrPadRight, padright 12
2.3.10 SysStrPadLeft, padleft 13
2.3.11 SysFmtCurrTime, strftimecurr 13
2.3.12 SysTime, time2epoch 14
2.3.13 SysStrFTime, strftime 16
2.3.14 SysInTable, intable 16
2.3.15 SysStrCondPack, condpack 17
2.3.16 TermAppStructDataGet, sfget 18
2.3.17 TermAppStructDataSet, sfset 19
2.3.18 gsub, replace 19
2.3.19 alias, lookup 21
2.3.20 uuid_time_gen 22
2.3.21 pstore_set, psset 22
2.3.22 pstore_get, psget 23
2.3.23 pstore_get_cset, psget_cset 24
2.3.24 pstore_get_incr, psget_incr 24
2.3.25 pstore_get_incr_cset, psget_incr_cset 25
2.3.26 runif 26
2.3.27 rnorm 27
2.3.28 rexp 27

List of Figures

1 Overview

1.1 Introduction

This guide explains how and when expressions are used and evaluated in Code Magus products that are enabled to use them. It also catalogues the list of operators, including their precedence, and built-in functions that may be used and presents comprehensive examples of using expressions.

The C programming API and more in-depth information can be found in the manual ‘ expeval: Expression Evaluation API Reference[1]’.

2 Expression Evaluation Syntax

2.1 Expression Overview

The lexical elements of an expression are the variables, literals, operators and other character symbols used to form an expression. These lexical elements or tokens are separated by white spaces. White spaces include sequences of the space character, new-line character, the tab character and the linefeed character and their only function is to separate or delimit the tokens.

The lexical elements are often single characters having their own apparent meaning, but some are grouped together to form a word having a specific meaning. Included or associated with each token may be an attribute value.

An expression, made up of the constituent tokens into the syntax and semantics of the grammar, is then validated and evaluated by the expression evaluation library. The evaluation of an expression produces a value that can then be used within the context of the grammar of the specific Code Magus product within which it is specified.

Examples of expressions are:

1. `3+4`
2. `balance + 100`
3. `(account.balance >= 2000)`
4. `where (account.balance = 0)`
5.
```
where (account.balance < 0 ) and
      (account.overdraft_facility = 'Y')
```
6. `SysString(account.balance)`

2.2 Expression Grammar

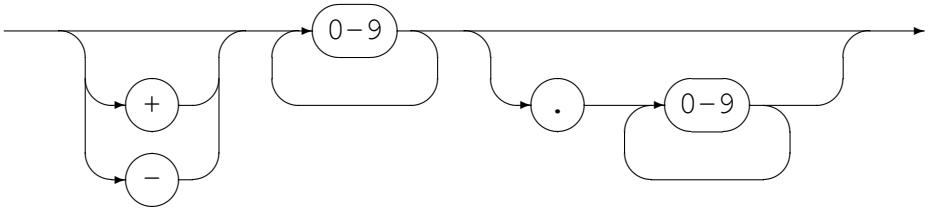
2.2.1 Lexical Elements

The base elements are *Literals* and *Identifiers*.

- Numeric Literals

  A Numeric literal is made up from an optional plus or minus sign followed by one or more digits and optionally followed by a point and one or more digits.

  Number Literal



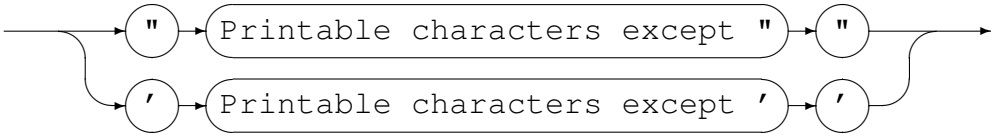
- String Literals

  String literals are made up from

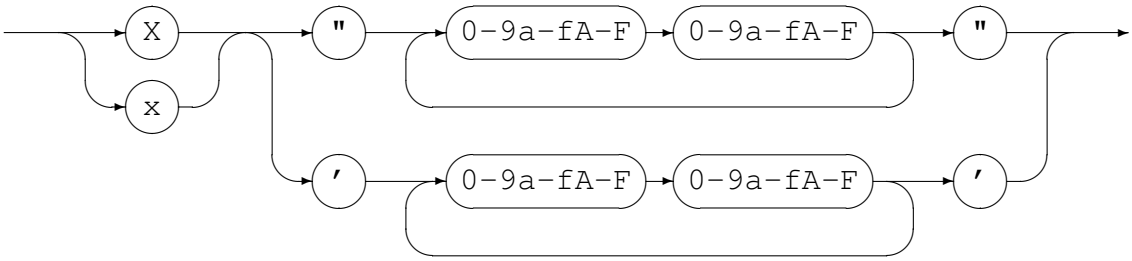
  – Any number of printable characters, except the enclosing character and a newline, enclosed in either single or double quotes.

  – An even number of hexadecimal digits enclosed in either single or double quotes and prefixed with a lower or upper case `X`.

  String Literal



  Hexadecimal Literal

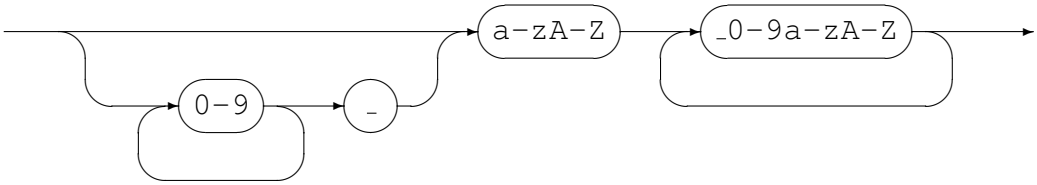


- Identifiers

  An identifier is used for both variable and function names. An identifier must conform to:

- A lower or upper case alphabetic character followed by any number of underscores, decimal digits and upper and lower case alphabetic characters.
- One or more decimal digits followed by an underscore and the above rule.

Identifier

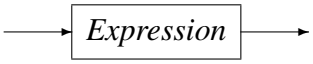


2.2.2 Syntactical Elements

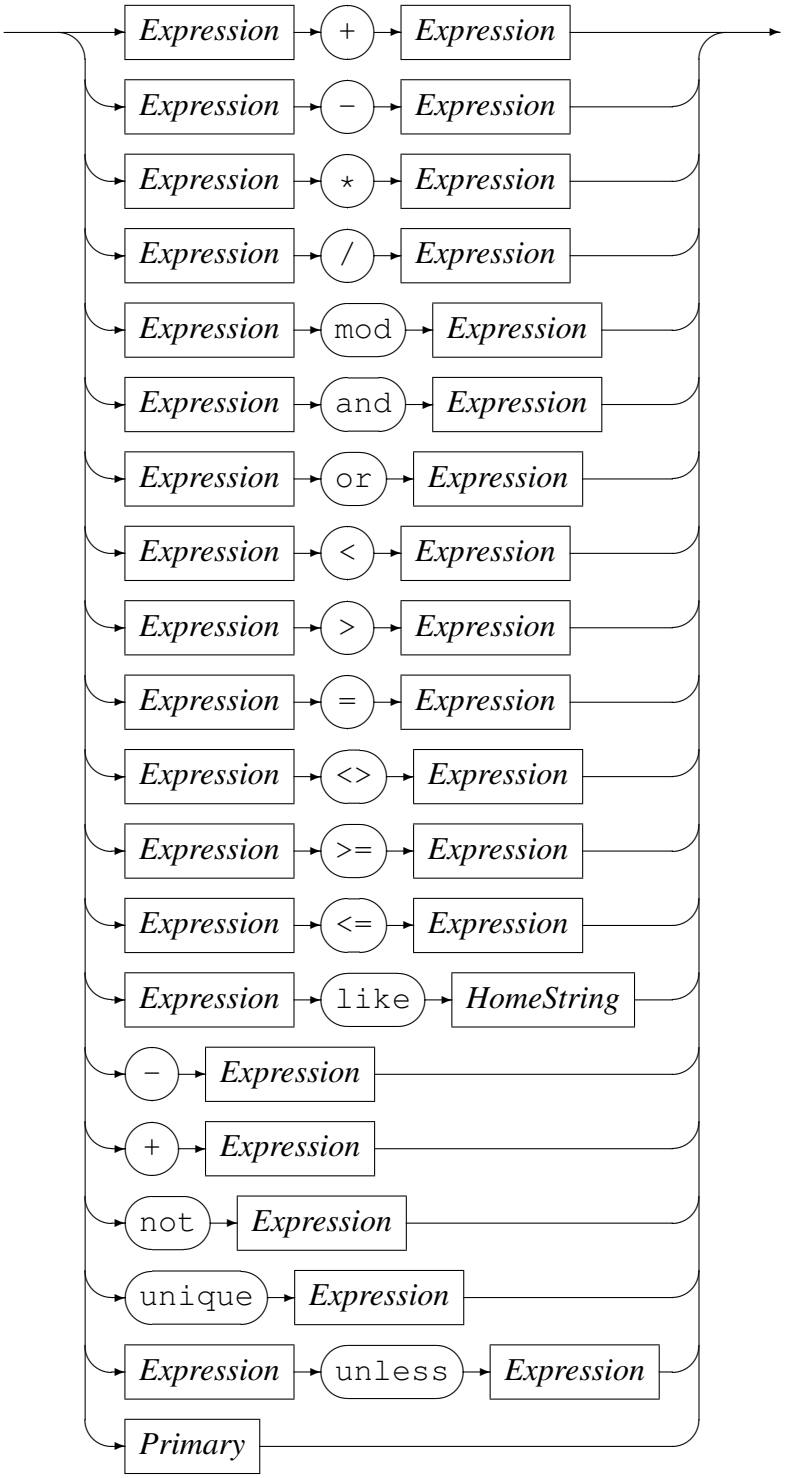
Expressions may themselves be used as syntactical elements when forming a compound expression.

The complete syntax of a compound expression is explained in the following sections starting with the compound expression and working down to the lowest level syntactic element.

CompoundExpression



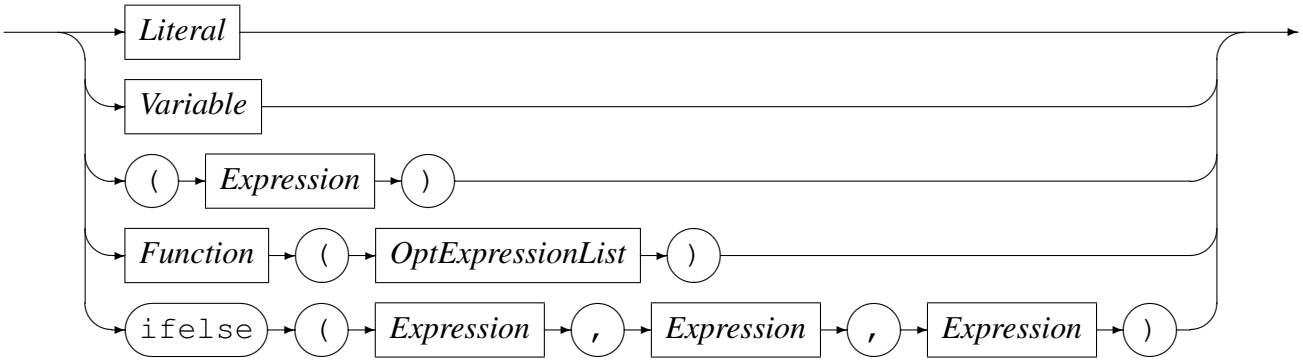
Expression



The `unless`-operator conditionally returns the value of the right-hand operand, unless there is an error evaluating the right-hand operand. In the case where the right-hand operand fails to evaluate to a proper value, the value of the left-hand operand is returned

instead. The left-hand operand is always evaluated before the right-hand operand. If the left-hand operand fails to evaluate to a proper value, then the result of the `unless`-operator is a failure.

Primary



As a terminal in the syntax structure an expression or *Primary* is either a *Literal* or a *Variable*, an *Expression* enclosed in parenthesis, a *Function* call reference, or the conditional evaluation operator `ifelse`. A *Literal* may be a *String Literal* or a *Number Literal* as described in Section 2.2.1 on page 3.

Where required by the encoding indicated or defaulted, characters representing the attribute value of a string are changed to an alternate character set if the required character set is not the same as the home character set being used. For example, on a machine in which the characters are naturally represented using the EBCDIC character set encoding (such as code page of 1047 or Latin 1/Open Systems), if the data being processed is from a machine in which the characters are naturally represented using the ASCII character set (such as ISO8859-1), then the characters in the String literal (assumed to be represented in EBCDIC) will be translated to their corresponding ASCII characters for processing. This does not apply to String literals that were represented as a sequence of hexadecimal digits.

Both a *Function* (see Section 2.2.2 on page 8) and an *Expression* are made up of sub-expressions, although eventually even they must terminate and resolve to a value.

A *HomeString* is a *String Literal* that may not be represented as a sequence of hexadecimal digits, but in which the encoding is left in the natural encoding of the machine processing the data; that is the machine on which the expression string is being compiled. This is required for the right-hand operand of the like operator as this operator translates the value of the left-hand operand into the local encoding when performing pattern matching.

Operators, variables and functions are described in more detail below:

- Operators

  In the context of the expression evaluation library, an operator is a symbol that

operates on or causes an action to be be performed on the constants and variables adjacent to it. An operator is either

- Monadic
  A monadic operator only operates on one value and usually employ either prefix or postfix notation in that they either occur before or after the value they operate on. The expression evaluation library uses only prefix monadic operators.
- Dyadic
  Dyadic operators operate on two values and employ infix notation in that they operate on the the values that immediately precede and follow the operator.

All operators return a value of a defined type which is the result of the computation. The type returned by an operator must be semantically consistent within the context of the rest of the expression and the grammar it may be embedded in.

Table 1 on page 7 lists the allowed operators, their precedence, associativity, arity (whether or not they are monadic or dyadic) and Type.

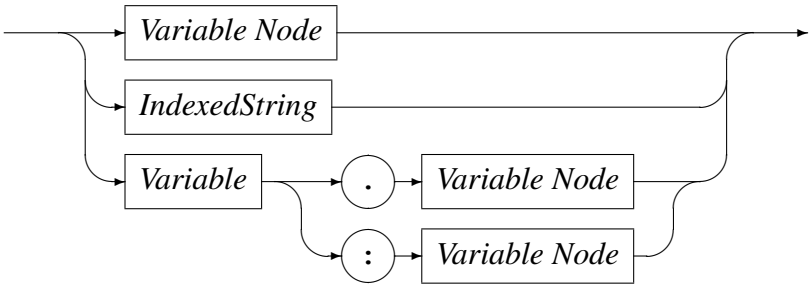
Operator	Precedence	Associativity	Arity	Type
`like`	1	non-assoc	dyadic	Relational
`<>`	1	left	dyadic	Relational
`>=`	1	left	dyadic	Relational
`<=`	1	left	dyadic	Relational
`=`	1	left	dyadic	Relational
`>`	1	left	dyadic	Relational
`<`	1	left	dyadic	Relational
`+`	2	left	dyadic	Arithmetic
`-`	2	left	dyadic	Arithmetic
`or`	2	left	dyadic	Boolean
`*`	3	left	dyadic	Arithmetic
`/`	3	left	dyadic	Arithmetic
`div`	3	left	dyadic	Arithmetic
`and`	3	left	dyadic	Boolean
`mod`	3	left	dyadic	Arithmetic
`-`	4	left	monadic	Arithmetic
`not`	4	left	monadic	Boolean
`unique`	4	left	monadic	boolean
`unless`	5	left	dyadic	boolean

Table 1: Operators: Precedence, Associativity, Arity and Type

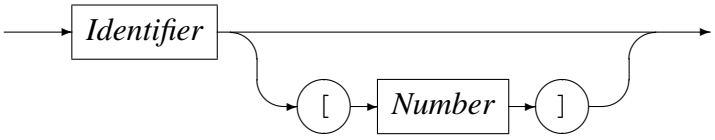
- Variables

A variable is the name of a storage location that holds a value. Simply this name is just an *Identifier*, but may be more than one level or node including an index.

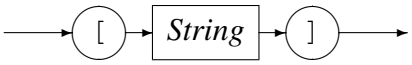
Variable



Variable Node



IndexedString



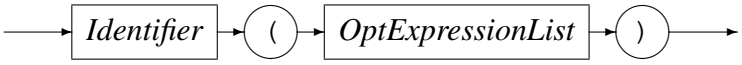
Examples of variable names are:

- `Address` - A single node variable with no indexing.
- `Customer.Address` - A two node variable.
- `Customer.Address[1]` - A two node variable where the Address portion of the variable is the first of an array of items. Here this may be the first line of an address.
- `Customer[3].Address[1]` - A two node variable that specifies the third entry of the Customer array and the first entry of the Address array within that Customer.
- `Customer.Contact.HomePhone` - A three node variable.

- Functions

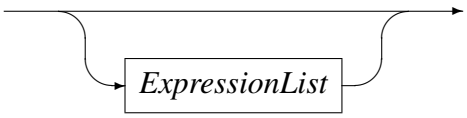
  A function is a special type of operator. It is specified by the function name, an identifier, followed by a comma separated list of arguments enclosed in parentheses.

  Function



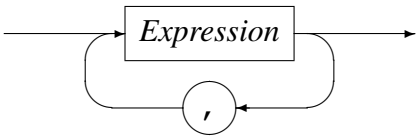
where an optional expression list is defined as

OptExpressionList



and an expression list is defined as

ExpressionList



The function call is replaced with the result of the call and the result type must be semantically consistent within the context of the rest of the expression and the grammar it may be embedded in.

2.3 Built-in Functions

Functions for expression evaluation can be supplied by the application that uses it and as such has a rich set of plug in functions that can not be documented here. However there are functions that are common to all data processing and these are supplied by the expression evaluation library and are described below.

2.3.1 SysStrLen, strlen, length

- **Synopsis**
  - `SysStrLen(string)`
  - `strlen(string)`
  - `length(string)`
- **Parameters**
  - Parameter 1 type: String.
- **Description**

  The `SysStrLne` function (aliases `strlen`, `length`) returns the number of characters in the string supplied as the first argument.

2.3.2 SysSubStr, substr

- **Synopsis**
  - `SysSubStr(string,start,length)`
  - `substr(string,start,length)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: Number.
  - Parameter 3 type: Number.
  - Return type: String.
- **Description**

  The `SysSubStr` function (alias `substr`) returns a substring of the given string from start for length characters or the remainder of string whichever is the shortest.

  The start must be greater than zero and the length must be zero or greater. If the start position is past the end of the string then a NULL string is returned.

2.3.3 SysString, string

- **Synopsis**
  - `SysString(number)`
  - `string(number)`
- **Parameters**
  - Parameter 1 type: Number.
  - Return type: String.
- **Description** The `SysString` function (alias `string`) returns the value of number as a string.

2.3.4 SysNumber, number

- **Synopsis**
  - `SysNumber(string)`
  - `number(string)`

- **Parameters**
  - Parameter 1 type: String.
  - Return type: Number.
- **Description** The `SysNumber` function (alias `number`) returns a number equivalent to the value of string.

2.3.5 SysStrCat, strcat

- **Synopsis**
  - `SysStrCat(first,second)`
  - `strcat(first,second)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** The `SysStrCat` function (alias `strcat`) returns a String which is the concatenation of the two input strings `first` and `second`.

2.3.6 SysStrStr, strstr

- **Synopsis**
  - `SysStrStr(haystack,needle)`
  - `strstr(haystack,needle)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** The `SysStrStr` function (alias `strstr`) returns the start position of `needle` within `haystack`. If `needle` does not occur in `haystack` then zero is returned, otherwise the position (origin 1) is returned.

2.3.7 SysStrSpn, strspn

- **Synopsis**
  - `SysStrSpn(string,accept)`
  - `strspn(string,accept)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** The `SysStrSpn` function (alias `strspn`) returns the number of characters (bytes) in the initial segment of `string` which consist only of characters from `accept`.

2.3.8 SysStrCspn, strcspn

- **Synopsis**
  - `SysStrCspn(string,reject)`
  - `strcspn(string,reject)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** The `SysStrCspn` function (alias `strcspn`) returns the number of characters (bytes) in the initial segment of `string` which do not match any character from `reject`.

2.3.9 SysStrPadRight, padright

- **Synopsis**
  - `SysStrPadRight(string,length,pad)`
  - `padright(string,length,pad)`
- **Parameters**
  - Parameter 1 type: String.

  - Parameter 2 type: Number.
  - Parameter 3 type: String. Although the type is String, only the first character is used as the pad character.
  - Return type: String.
- **Description** The `SysStrPadRight` function (alias `padright`) returns a string whose length is `length` and:
  - if `length` is greater than the length of `string`, is `string` padded on the right with the `pad` character
  - if `length` is less than the length of `string`, is `string` truncated from the right to `length`.
  - if `length` is equal to the length of `string`, is `string`.

2.3.10 SysStrPadLeft, padleft

- **Synopsis**
  - `SysStrPadLeft(string,length,pad)`
  - `padleft(string,length,pad)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: Number.
  - Parameter 3 type: String. Although the type is String, only the first character is used as the pad character.
  - Return type: String.
- **Description** The `SysStrPadLeft` function (alias `padleft`) returns a string whose length is `length` and:
  - if `length` is greater than the length of `string`, is `string` padded on the left with the `pad` character
  - if `length` is less than the length of `string`, is `string` truncated from the left to `length`.
  - if `length` is equal to the length of `string`, is `string`.

2.3.11 SysFmtCurrTime, strftimecurr

- **Synopsis**

  - `SysFmtCurrTime(format)`
  - `strftimecurr(format)`
- **Parameters**
  - Parameter 1 type: String.
  - Return type: String.
- **Description** The `SysFmtCurrTime` function (alias `strftimecurr`) returns a string that represents the current time as formatted according to `format` using the C run-time `strftime()` function. Common values for `format` are:
  - %c - The preferred date and time representation for the current locale.
  - %d - The day of the month as a decimal number (range 01 to 31).
  - %F - Equivalent to %Y-%m-%d (the ISO 8601 date format).
  - %H - The hour as a decimal number using a 24-hour clock (range 00 to 23).
  - %j - The day of the year as a decimal number (range 001 to 366).
  - %m - The month as a decimal number (range 01 to 12).
  - %M - The minute as a decimal number (range 00 to 59).
  - %s - The number of seconds since the Epoch, 1970-01-01 00:00:00
  - %S - The second as a decimal number (range 00 to 60, allows for leap seconds).
  - %T - The time in 24-hour notation (%H:%M:%S).
  - %y - The year as a decimal number without a century (range 00 to 99).
  - %Y - The year as a decimal number including the century.
  - %% - A literal '%' character.
  - Any other characters, not specified by `strftime()`, are copied verbatim from `format` to the result string.

2.3.12 SysTime, time2epoch

- **Synopsis**
  - `SysTime(datetime,format)`
  - `time2epoch(datetime,format)`
- **Parameters**

- Parameter 1 type: String.
- Parameter 2 type: String. Default "%Y%m%d".
- Return type: Number.

- **Description** The `SysTime` function (alias `time2epoch`) returns the number seconds since the Epoch calculated from `datetime` under the specification of `format`.

  The seconds since the Epoch, when interpreted as an absolute time value, represents the number of seconds elapsed since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`.

  `datetime` must be a string representation of a date and / or time and `format` must be a date format string that exactly describes `datetime` using the format characters as specified and used by the C function `strptime()`.

  Common options for the `format` are:

  - %% - The % character.
  - %c - The date and time representation for the current locale.
  - %C - The century number (0-99).
  - %d or %e - The day of month (1-31).
  - %H - The hour (0-23).
  - %I - The hour on a 12-hour clock (1-12).
  - %j - The day number in the year (1-366).
  - %m - The month number (1-12).
  - %M - The minute (0-59).
  - %p - The locale's equivalent of AM or PM. (Note: there may be none.)
  - %S - The second (0-60; 60 may occur for leap seconds; earlier also 61 was allowed).
  - %T - Equivalent to %H:%M:%S.
  - %x - The date, using the locale's date format.
  - %X - The time, using the locale's time format.
  - %y - The year within century (0-99). When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969-1999); values in the range 00-68 refer to years in the twenty-first century (2000-2068).

- %Y - The year, including century (for example, 1991).

2.3.13 SysStrFTime, strftime

- **Synopsis**
  - `SysStrFTime(seconds,format)`
  - `strftime(seconds,format)`
- **Parameters**
  - Parameter 1 type: Number.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** The `SysStrFTime` function (alias `strftime`) returns a string date time representation of `seconds` formatted according to `format` as described in the C runtime function `strftime()`.

  `seconds` is the number of seconds since the Epoch, which when interpreted as an absolute time value, represents the number of seconds elapsed since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`.

  `format` must be a date format string used to format the returned date time string. For common values of `format` see section 2.3.11 on page 14

2.3.14 SysInTable, intable

- **Synopsis**
  - `SysInTable(table,search)`
  - `intable(table,search)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Boolean.
- **Description**

  The `SysInTable` function (alias `intable`) returns a boolean `TRUE` if the value of `search` is found in the table of items `table`, otherwise it returns a boolean `FALSE`.

The value of `table` may be either the name of a text file in which each line is one element of the table, or a comma (,) or semi-colon (;) delimited string of the element values of the table.

- **Examples**
  - SysInTable(”C:\customerNames.txt”,”Smith”) This will test whether the name ”Smith” occurs in the list of elements in the file `C:\customerNames.txt`.
  - SysInTable(”/tmp/customerNames.txt”,Record.Surname) This will test whether the name identified by the object types[2] field `Record.Surname` occurs in the list of elements in the file `/tmp/customerNames.txt`.
  - SysInTable(”Smith,Jones,Right”,Record.Surname) This will test whether the name identified by the object types[2] field `Record.Surname` occurs in the list of elements in the comma separated list specified by the first argument.

2.3.15 SysStrCondPack, condpack

- **Synopsis**
  - `SysStrCondPack(String,String)`
  - `condpack(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description**

  The `SysStrCondPack` function (alias `condpack`) returns a string which is conditionally formed by packing the string passed in the first parameter using the second parameter as a possible replacement character. If the first parameter matches the regular expression `X"[0-9][A-F][a-f]"` then the hexadecimal characters are packed into the corresponding encoding character set (ASCII or EBCDIC) characters. If the second parameter does not have a zero length, then the first character of this parameter string is used to replace all the non-graphic/non-printable characters of the packed character string. When the second parameter string has a zero length, then the character ”?” is used as the replacement character for non-graphic/non-printable characters in the return string.

  If the first parameter string does not match the regular expression then the string is considered to already be packed. In this case, the string is still checked if the

second parameter length is greater than one and the non-graphic/non-printable characters are replaced by the first character of the second parameter string. When the second parameter string has a zero length, then the character "?" is used as the replacement character for non-graphic/non-printable characters in the return string.

- **Examples**
  - `condpack('X"414141"',"?")` on an ASCII based machine returns the string `AAA`.
  - `condpack('X"4141410000"',"?")` on an ASCII based machine returns the string `AAA??`.
  - `condpack("4141410000","?")` on an ASCII or EBCDIC based machine returns the string `4141410000`.

2.3.16 TermAppStructDataGet, sfget

- **Synopsis**
  - `TermAppStructDataGet(String,String)`
  - `sfget(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function takes as the first parameter a value that should contain a `TermApp DE48-F0.16` Structured Data field and as the second parameter the name of a field within the structured data. The function will return the value of the named field as a string, if the name could not be found an empty string is returned.
- **Examples**
  - `sfget(DE48_FIELD,"OSVer")`
    Where**DE48_FIELD=**
    `219Postilion::MetaData275211FWSerialNbr11115SWRel111 19CommsType11118TermType11115OSVer11116SWHash111211F 01E201WSerialNbr2210100010000000100224231 5SWRel21314 4060219CommsType214INTERNAL MODEM 18TermType18EFTsma rt15OSVer198200360781 6SWHash18B4E1963A`

returns the string `820036078`

2.3.17 TermAppStructDataSet, sfset

- **Synopsis**
  - `TermAppStructDataSet(String,String,String)`
  - `sfset(String,String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.
- **Description**
- **Examples**
  - `sfset(DE48_FIELD,"FWSerialNbr",`
    `"+----------LongerValue----------+")`

    Where **DE48_FIELD** is initially set to
    `219Postilion::MetaData275211FWSerialNbr11115SWRel111`
    `19CommsType11118TermType11115OSVer11116SWHash111211F`
    `WSerialNbr2210100010000000100224231 5SWRel2131401E201`
    `4060219CommsType214INTERNAL MODEM18TermType18EFTsmar`
    `t15OSVer1982003607816SWHash18B4E1963A`

    Will return the updated value of **DE48_FIELD** as
    `219Postilion::MetaData275211FWSerialNbr11115SWRel111`
    `19CommsType11118TermType11115OSVer11116SWHash111211F`
    `WSerialNbr233+----------LongerValue----------+15SWRe`
    `l2131401E2014060219CommsType214INTERNAL MODEM18TermT`
    `ype18EFTsmart15OSVer1982003607816SWHash18B4E1963A`

2.3.18 gsub, replace

- **Synopsis**
  - `gsub(String,String,String,String)`
  - `replace(String,String,String,String)`

- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Parameter 4 type: String.
  - Return type: String.
- **Description** The function gsub() operates in much the same why as the `awk gsub` function does. The four parameters are
  1. Regular Expression (r) This parameter is a regular expression that should match one or more portions of the input text (t).
  2. Substitution String (s) This parameter is the replacement string
  3. Text to operate on (t) This parameter is the original input text value.
  4. How to operate (h) This parameter determines how many times the replacement text is substituted.

  How (h) can be either
  - g or G which means replace all occurrences of matched text with the substitution string.
  - Numeric which means replace only that occurrence.

  The regular expression (r) matches none, one or more portions of the input text (t) and based on the value of how (h) gsub() returns the input string where one or all of the matches are replaced with the substitution string (s).
- **Examples**
  - `gsub("a","bb",textfield,how)` This example specifies to replace the letter `a` with two letter `b`'s in `textfield` under the control of the variable `how`.

Textfield value	How	Returned Value	Description
abcdea12345a	G	bbbcdebb12345bb	Each a is replaced by two b's.
abcdea12345a	2	abcdebb12345a	The second a is replaced by two b's.
abcdea12345a	1	bbbcdea12345a	The first a is replaced by two b's.

Table 2: Effect of using gsub() to substitute text

  - `gsub("\([^ ]\+\) \([^ ]\+\)","\2 \1",textfield,how)` This example specifies to match two substrings that contain any character

except a space and that the first substring must be followed by a space followed by the second substring. The substitution string specifies to replace the whole matched value with the second matched substring followed by a space followed by the first matched substring. In other words it swaps two substrings around where the substrings do not contain a space and are separated by one space. The number of times the replacement is done is governed by the value of the variable `how`.

Textfield value	How	Returned Value	Description
ABC DEF	G	DEF ABC	The order of the two strings is reversed.
A1 bA1 A2 BA2	G	bA1 A1 BA2 A2	Each set of two strings are reversed.
A1 bA1 A2 BA2	2	A1 bA1 BA2 A2	Only the second set is reversed.

Table 3: Effect of using gsub() to substitute text

2.3.19 alias, lookup

- **Synopsis**
  - `alias(String,String)`
  - `lookup(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function uses the second parameter as a lookup key to extract the associated value in the first parameter, which holds keyword value pairs. The value corresponding to the matched key word is returned. The keyword value pairs specified in the first parameter can either be a comma or semi-colon list of `keyword=value` pairs or a file name containing one `keyword=value` pair per line.
- **Examples**
  - `lookup("A=Alsatian,L=Labrador,S=Spaniel","L")`
    Will return the string ”Labrador”
  - `lookup("D:/lookup.txt","L")`
    will return the string ”Labrador” if the file `D:/lookup.txt` holds the following:

```
A=Alsatian
L=Labrador
S=Spaniel
```

2.3.20 uuid_time_gen

- **Synopsis**
  - `uuid_time_gen()`
- **Parameters**
  - Function takes no parameters.
- **Description** The function uuid_time_gen() uses the `libuuid` library (`https://sourceforge.` to generate a time based UUID using the system's local clock and network interface MAC address (if available). Each time the function is called it returns a new value which is expected to be globally unique.
- **Examples**
  - The sequence of calls to the `uuid_time_gen()` function returns a UUID formatted string:
  - `uuid_time_gen()` returns `a61293e0-58fe-11e8-97e1-f9f10e0b5eac`.
  - `uuid_time_gen()` returns `a6129462-58fe-11e8-97e1-f9f10e0b5eac`.
  - `uuid_time_gen()` returns `a612949e-58fe-11e8-97e1-f9f10e0b5eac`.
  - `uuid_time_gen()` returns `a61294d0-58fe-11e8-97e1-f9f10e0b5eac`.

2.3.21 pstore_set, psset

- **Synopsis**
  - `pstore_set(String,String,String)`
  - `psset(String,String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.

- **Description** This function sets a value in a persistent store specified in parameter 1 using the variable name specified in parameter 2 and the value in parameter 3. If an error occurs, for example not being able to connect to the persistent store server, an error condition is returned.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.

- **Examples**
  - `pstore_set("www.codemagus.com:60069","ServerName","theCloud")` Will set and return the value of the variable `ServerName` to `theCloud` on the specified host.
  - `psset("www.codemagus.com:60069","ServerName","theCloud")` Will perform the same function as the example above.

2.3.22 pstore_get, psget

- **Synopsis**
  - `pstore_get(String,String)`
  - `psget(String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: String.
- **Description** This function retrieves a value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. If the named variable is not found then an error condition is returned.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.

- **Examples**
  - `pstore_get("www.codemagus.com:60069","ServerName")` Will return the value of the variable `ServerName` from the specified host.

- `psget("www.codemagus.com:60069","ServerName")`
  Will perform the same function as the example above.

2.3.23 pstore_get_cset, psget_cset

- **Synopsis**
  - `pstore_get_cset(String,String,String)`
  - `psget_cset(String,String,String)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: String.
  - Return type: String.
- **Description** This function retrieves a value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. If the named variable is not found then it is created with the default value specified in parameter 3 and that value is returned.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.
- **Examples**
  - `pstore_get_cset("www.codemagus.com","ServerName","theNet")`
    Will return the value of the variable `ServerName` from the specified host (using the default port), but if it is not found will return and set it to `theNet`.
  - `psget_cset("www.codemagus.com","ServerName","theNet")`
    Will perform the same function as the example above.

2.3.24 pstore_get_incr, psget_incr

- **Synopsis**
  - `pstore_get_incr(String,String)`
  - `psget_incr(String,String)`

- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Return type: Number.
- **Description** This function retrieves a string representation of a numeric value from a persistent store specified in parameter 1 using the variable name specified in parameter 2. The numeric string is returned as a number type and is subsequently incremented by 1 and saved back to the persistent store as a numeric string.

  The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.
- **Examples**
  - `pstore_get_incr("www.codemagus.com:60069","Count")`
    If the value of `Count` on the persistent store is 3, then this function will return 3 and store 4 back on the persistent store. If the variable `Count` is not found an error condition is returned.
  - `psget_incr("www.codemagus.com:60069","Count")`
    Will perform the same function as the example above.

2.3.25 pstore_get_incr_cset, psget_incr_cset

- **Synopsis**
  - `pstore_get_incr_cset(String,String,Number)`
  - `psget_incr_cset(String,String,Number)`
- **Parameters**
  - Parameter 1 type: String.
  - Parameter 2 type: String.
  - Parameter 3 type: Number.
  - Return type: Number.
- **Description** This function retrieves a string representation of a numeric value from a persistent store specified in parameter 1 using the variable name specified

in parameter 2. The numeric string is returned as a number type and is subsequently incremented by 1 and saved back to the persistent store as a numeric string. If the named variable is not found on the persistent store then the default value specified in parameter 3 is returned and subsequently incremented and saved on the persistent store.

The persistent store is identified by `host:port` where `host` is either an IP address or a DNS name that can be looked up and `port` is the port number on that host to which the persistent store server listens for incoming connections. The port (and the colon) can be left out in which case the default port is used. The default port is currently 60060.

- **Examples**
  - `pstore_get_incr_cset("codemagus","Count",17)`
    If the value of `Count` on the persistent store is 3, then this function will return 3 and store 4 back on the persistent store. If the variable `Count` is not found then the value 17 is returned and 18 is saved to the persistent store as the value of `Count`.
  - `psget_incr_cset("codemagus","Count",17)`
    Will perform the same function as the example above.

2.3.26 runif

- **Synopsis**
  - `runif(Number,Number)`
- **Parameters**
  - Parameter 1 type: Number. Minimum value.
  - Parameter 2 type: Number. Maximum value.
  - Return type: Number.
- **Description** This function returns a uniform random number/deviate between the minimum value (first argument) and the maximum value (second argument). The minimum value must be strictly less than the maximum value. The random numbers returned are real numbers.
- **Examples**
  - `runif(1,100)` The call to function `runif` in this examples returns a uniform random real number in the range 1 to 100.

2.3.27 rnorm

- **Synopsis**
  - `rnorm(Number,Number)`
- **Parameters**
  - Parameter 1 type: Number. Mean value.
  - Parameter 2 type: Number. Standard deviation.
  - Return type: Number.
- **Description** This function returns a random/deviate number from a normal distribution with parameters supplied by the first and second parameters. The value supplied by the first argument is the population mean and the value supplied by the second argument is the population standard deviation. The standard deviation must be strictly greater than zero. The normal random number returned and the supplied mean value and standard deviation are real numbers.
- **Examples**
  - `rnorm(100,10)` The call to function `rnorm` in this examples returns a normal random real number with mean value if 100 and standard-deviation of 10.

2.3.28 rexp

- **Synopsis**
  - `rexp(Number)`
- **Parameters**
  - Parameter 1 type: Number. The mean value of the exponential distribution.
  - Return type: Number.
- **Description** This function returns a random number/deviate drawn from a negative exponential distribution. The value of the mean is supplied by the first and only parameter. The returned value and the value of the mean are real numbers.
- **Examples**
  - `rexp(20)` The call to function `rexp` in this examples returns an exponential random real number with mean value 20.

References

[1] expeval: Expression Evaluation API Reference. CML Document CML00052-01, Code Magus Limited, July 2024. PDF.

[2] Code Magus Limited. objtypes: Configuring for Object Recognition, Generation and Manipulation. CML Document CML00018-01, Code Magus Limited, July 2008. PDF.