CML Query V2: Web Query Tool Version 2

CML00118-01

Code Magus Limited (England reg. no. 4024745)
Number 6, 69 Woodstock Road
Oxford, OX2 6EY, United Kingdom
www.codemagus.com

November 23, 2020

Contents

List of Figures

1 Introduction

The Code Magus Limited Query System is a web enabled data query system that offers fast searching over a wide area of the data, as well as deeper targeted searches to answer specific queries. Users, such as a data analyst, can search and interrogate the data within an application domain to which they are granted access.

Chronological (or time series) transaction logs are excellent candidates for these types of queries.

2 Overview

A user can query data simply by supplying search patterns for up to 10, pre-configured key fields and an optional time range. The key fields are chosen for the application such that using them should cater for nearly all the queries a data analyst would normally need to perform. See image 1 on page 4 for an example of a Credit Card Logging application domain. If a user wishes to examine all credit card transactions for a particular POS (point of sale) terminal in a set of transaction logs they just select the correct application domain and enter the card number (or part of it) in the `Primary Account Number` key input field.

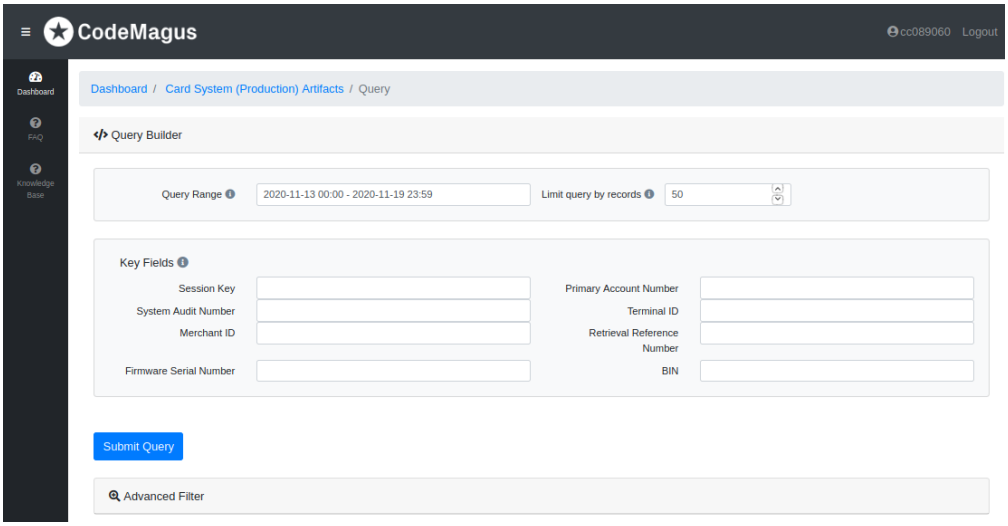


Figure 1: Query Submit Form

Should a more in depth query be required, the Query System also exposes an advanced filter using meta data and expressions over the meta data as explained in objtypes: Configuring for Object Recognition, Generation and Manipulation[2] and expeval: Expression Evaluation User Guide[1] or appendix A on page 11. The user can construct very detailed query configurations. These advanced filter configurations can be saved for future use. Figure 2 on page 5 shows an example of an advanced filter.

The Query System is designed to return relevant data as quickly as possible. The initial result set screen displays control information (record type, associated groups and timestamp), the key fields and up to ten pre-configured fields of interest. All key fields are links that if clicked on will return a result set of all records (within the current date range and maximum rows set) that match the index value. Each returned record can be further expanded to display all fields in the record should a user wish to view them. Figure 3 on page 5 shows an example of a result from a query.

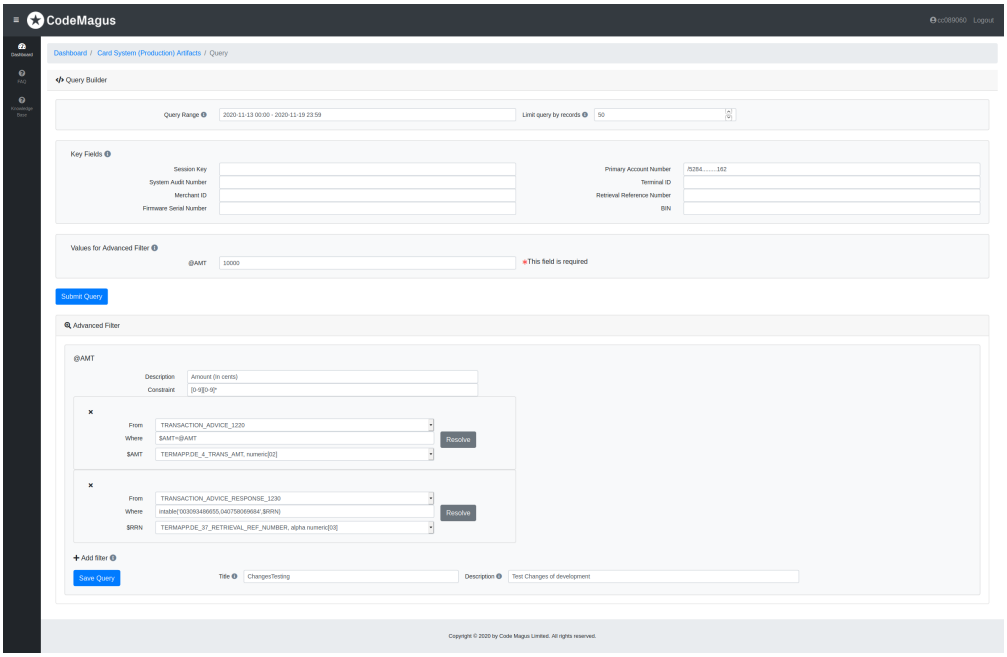


Figure 2: Query with Advanced Filter

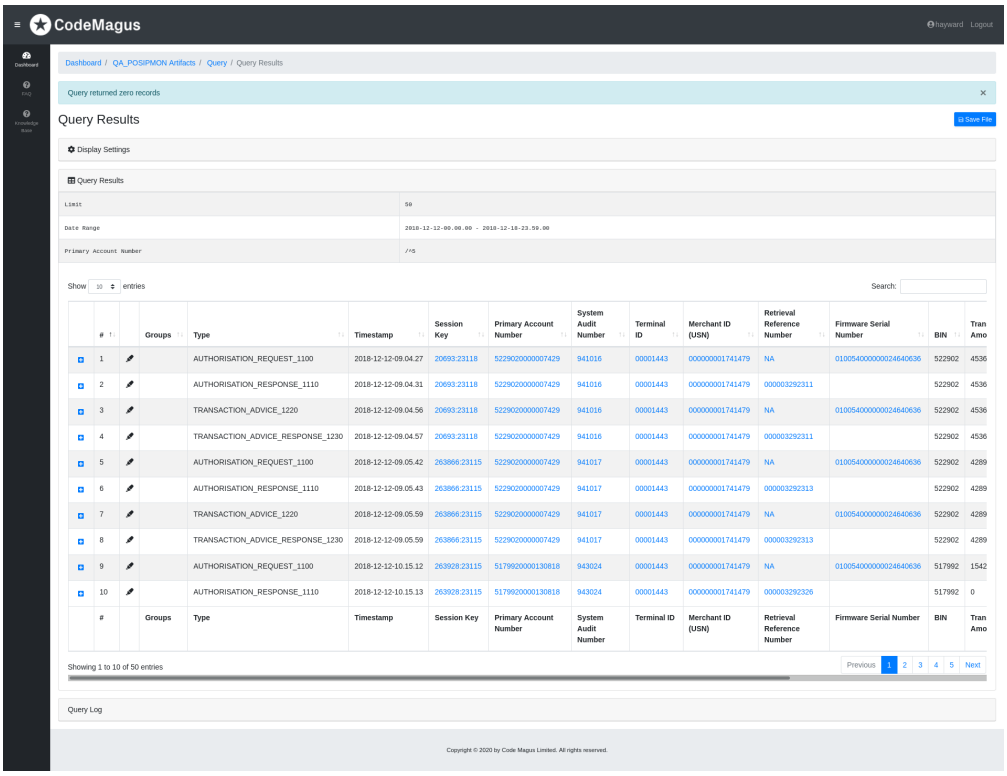


Figure 3: Query Results

3 User Guide

3.1 The Key Fields

The key fields are pre-defined per domain and there may be up to 10 of them. They provide a fast way to query the domain data as they are database indexes optimised for speed. All the key fields are treated as plain text even if the value they hold is numeric. This means that a search is done using text values.

There are three types of value that can be entered into any of the key fields and each will match the data (or not) in a slightly different way.

1. Simple match.

   By entering a value in a key field input text box, records will be filtered for records where the key field exactly matches this value. This does mean that a search value such as `"1234"` will not match a value `"AB1234"`.

2. Simple pattern.

   The two meta-characters `"_"` (underscore) and `"%"` (percent) will match respectively any single character and zero or more characters. Taking the example above `"%1234"` will match the value `"AB1234"`, but `"_1234"` will not.

3. Extended Regular expression pattern.

   To search using an extended regular expression, the pattern should start with a `"/"` (forward slash). The Query System will remove the first character and use the rest of the pattern as an extended regular expression.

   An extended regular expression is a pattern made up of actual characters and a rich number of meta-characters that make complex matching possible. Any (printable) character will match itself as in the two matching types above. A meta-character allows matching any value, ranges of values or sequences of the same item. Traditionally, all meta-characters are preceded by an escape character `"\"` (backslash) in order to identify them, but in extended regular expressions some meta-characters (for example `".*[]"`) do not need to be escaped. In extended regular expressions, for example, if matching an actual `"."` (full stop) then it needs to be escaped as `"\."`. For a good explanation of regular expressions and in particular extended regular expressions, refer to https://en.wikipedia.org/wiki/Regular_expression.

   Taking the example above, the expression `"/.*1234"`, means match any number of characters (including zero) followed by the value `"1234"` and will match the value `"AB1234"`.

   Most importantly the pattern `"/1234"` will match the value `"AB1234"`, as a regular expression matches any part of the value unless explicitly coded to not.

The meta-character "^" (Circumflex accent) matches the beginning of the value and "$" matches the end of the value. Therefore the pattern "/^1234$" will not match the value "AB1234".

3.2 The Advanced Filter

The advanced filter enables finely tuned and specific queries to be performed when using the Key fields returns too many false positives. For example, if in a card application the Primary Account Number (PAN) shown on a transaction slip has the middle digits masked with dots and querying on this using a regular expression in the key field would return many more rows than the single transaction required. A user could then use the advanced filter to limit the transaction set to only those with a specific amount.

To activate, click on ”Advanced Filter” at the bottom of the Query Builder page. From here you will need to add one or more filters by clicking on the plus sign to the left of ”Add filter”.

This opens a small dialogue (one for each filter added or one per object type selected) that allows the user to select from a drop down the ”From” object type to filter, and a text entry box in which to type the query expression.

The Query expression must be an expression that returns a Boolean TRUE or FALSE when executed against each record of the selected object type. At its simplest an expression could be `$PAN like '^542789.*1079'`.

Even this simple expression needs some explaining.

1. The literal `'^542789.*1079'`

   This literal is a regular expression pattern that will match a character field (although it is numeric data) referred to as `$PAN` (see below) where it starts with the given digits 542789, followed by anything (including nothing), followed by the digits 1079.

2. `$PAN` The variable reference.

   This could have been written as `TERMAPP.DE_2_PRIMARY_ACCOUNT_NUMBER`, as this is the fully qualified name for the Primary Account Number within this object type. But, that is difficult to remember, as this field may occur in many object types across many different application domains, and it may not always be the same name. So instead the `$` prefixed name of your own choice creates a reference variable. Here `PAN` is used as a short easy to remember acronym. Once the expression is typed in, click on the ”Resolve” button and a new drop down will appear with the title `$PAN` to the left. This is a list of all the fields in the type selected above. Once the correct name is selected, then `$PAN` will be associated with this name and correctly substituted in the query at execution time.

The query can be saved by entering a name and description and clicking on the ”Save Query” button. Note that any values entered into the key fields or advanced query prompts are not saved.

Saving a query is useful in the circumstance where it needs to be executed many times over. The above simple query, though, is not so useful if on subsequent days a user needs to run a query on a different PAN. For this, a query is needed where the value to query is prompted for each time. To achieve this change the simple query to look like this:

```
$PAN like '@pan'
```

When the ”Resolve” button is pressed now, the system defines a global prompt variable called @pan and adds two input text fields to the top of the advanced filter dialogue. It requires a ”Description” and a ”Constraint”. These values are used at query execution time to help the user running the query, when they will need to supply the substitution value for the pan. Remember that the person defining a query may not be the the person running it in the normal course of their work. The description should say what this prompt is asking for and the constraint is a regular expression that limits the input values that the user can key in. For example for a prompt of the PAN value, the description could be "Enter a Primary Account Number Mask" and the constraint should be a regular expression that only allows digits and trailing spaces, such as "^[0-9.]\+[ ]*", which means the value must start with one or more digits or dots (to represent a single digit), followed by zero or more spaces. The following spaces are not necessary here, but are used to show a more complex regular expression. If this query is saved, then each time it is selected to be executed, it will prompt the user for the value to search for. At its simplest this could just be the exact card number.

To create a query that searches for cards matching a pattern and a specific amount use an expression like this: ($PAN like '@pan') and ($AMOUNT = @amount) Associate $PAN and $AMOUNT with the correct fields in the drop down list of fields for the corresponding selected type and add a description and constraint to each of the global prompt fields shown at the top of the Advanced filter.

If this same query is required for more than one type, then add more filters and paste the same query in and click on the ”Resolve” button next to that query. Two more drop downs for $PAN and $AMOUNT will appear so that they can be associated to the correct field for this type. No more global prompt values will appear, as they have already been defined and the values entered by the query user will be substituted in for each additional filter.

It is highly likely that PAN (and possible AMOUNT) are declared as key fields. In this case there is no need to use them in the advanced filter as the key field value will be able to subset the data based on the value supplied there.

These simple expression have shown a couple of operands.

1. `like` for matching using a regular expression.
2. = for numeric equality.
3. `()` for grouping expressions.
4. `and` for logical and of two Boolean values.

There are many more expressions and built in functions that can be used. For more information please consult the expeval documentation in appendix A on page 11.

3.3 Query Performance

The speed with which the Query System returns records from the domain data depends on which type of key value or advanced filter is used.

The simple match is always the fastest and it is best to use as many key values as possible.

If a wild card search is necessary, then try putting the wild card portion towards the end of the value. The shorter the fixed portion at the front is, the slower the query will be. For example the following are ranked from the fastest to the slowest.

- `12345_` (Fastest)
- `12345%`
- `1234%6789`
- `12%45`
- `%12345` OR `/.*12345$`
- `%12345%` OR `/12345` (Slowest)

Note (as explained above in section 3.1 on page 6) that `/12345` is not the same as `12345` as the first is a regular expression and as such will search for `12345` anywhere in the data, and the second is an exact match; the data must start with `1` and end with `5`.

3.3.1 Effect of Row Limit on a Query

On the query input form the field '`Limit query by records`' is an upper limit of records to return. However, in terms of query performance it is most useful when there is no advanced filter specified. In this case, the Query System adds a limit clause to the database query. This is well optimised as the query itself will not return more than the requested amount of rows from the underlying database manager.

If, on the other hand there is an advanced filter, then the Query System does not add a limit clause to the database query as it needs to further qualify the results with the

advanced filter. For example for a limit of 50, if the values supplied would cause a very large result set to be returned, say 10000, the Query System has to wait until this is complete and it has all 10000 rows before applying the advanced filter to them and only applying the limit test to those that are selected by the advanced filter and returning only the first 50 of those. In other words in this example the system would have spent time searching for and returning (potentially) 9950 rows that are then discarded.

3.3.2 Performance Summary

In summary, try to follow these steps when querying the domain data:

1. Always try to use full index values for the best query performance.
2. If this is not possible, try to limit the wild card values required and have them towards the end of a value.
3. Use the advanced query with caution. Select a narrow date range or use it in conjunction with specific filter values.

A Lexical Elements, Syntax and Semantics

A.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)`

A.2 Expression Grammar

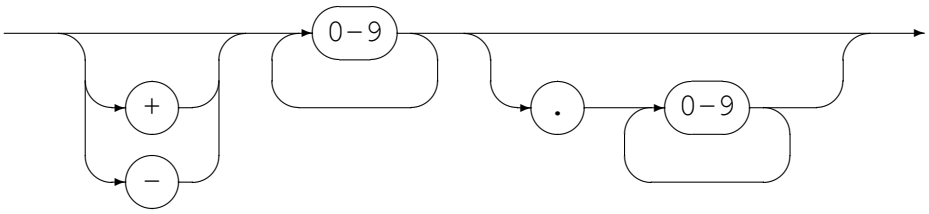
A.2.1 Lexical Elements

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



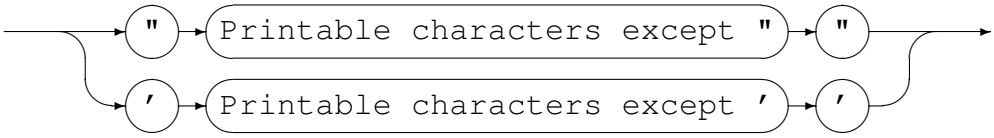
- String Literals

  String literals are made up from

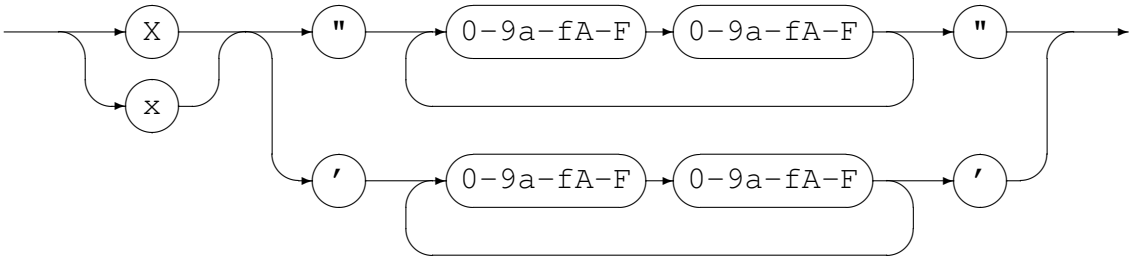
  – Any number of printable characters, except the enclosing character and a newline, enclosed in either single or double quotes.

  – 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



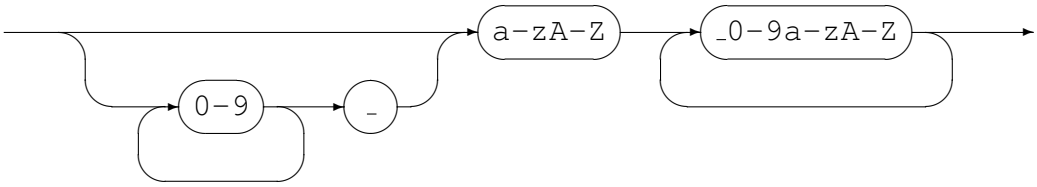
- Identifiers

  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

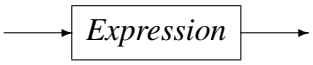


A.2.2 Syntactical Elements

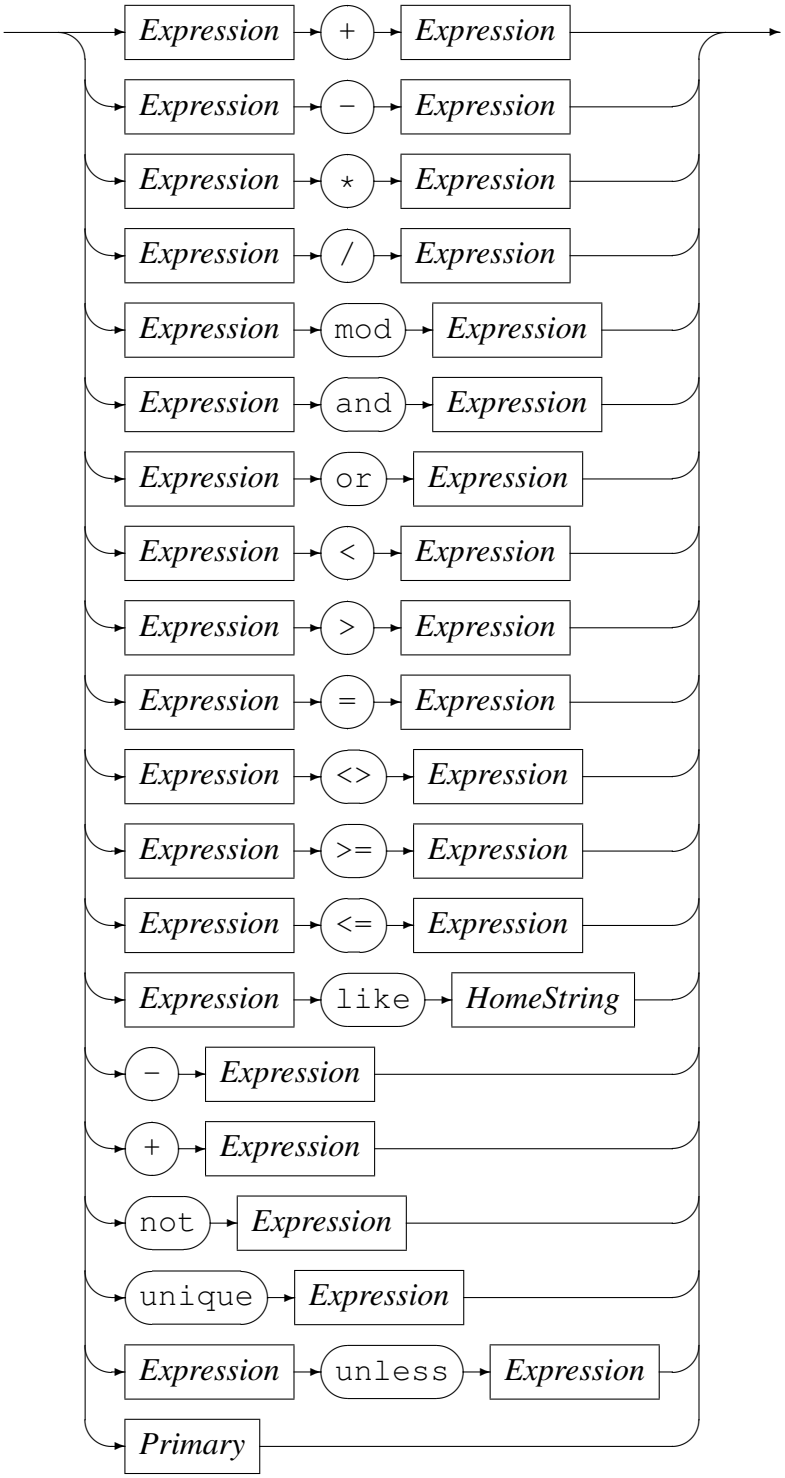
Expressions may themselves be used as syntactical elements when forming a compound expression.

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



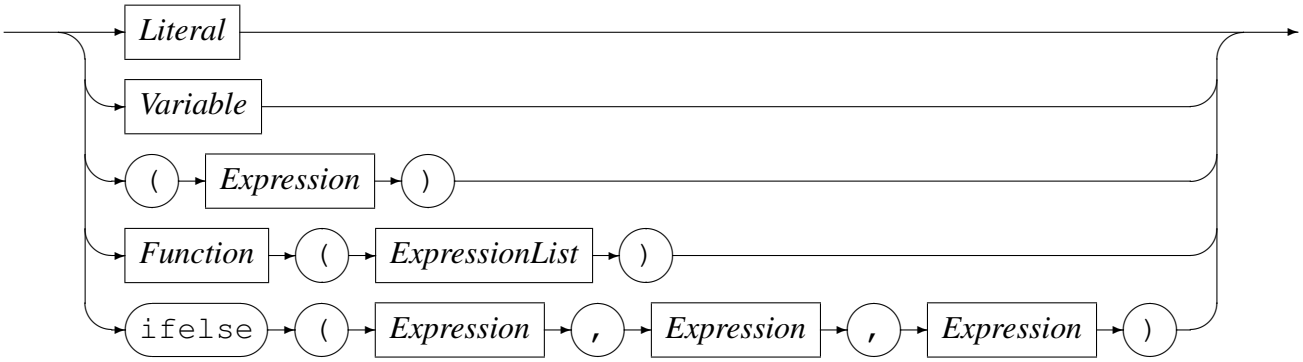
Expression



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 A.2.1 on page 11.

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 A.2.2 on page 17) 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 16 lists the allowed operators, their precedence, associativity, arity (whether or not they are monadic or dyadic) and Type.

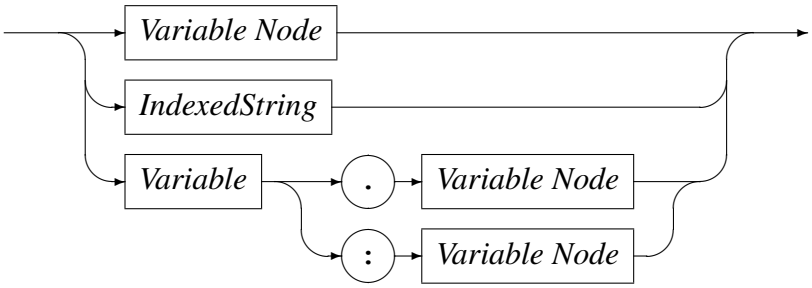
Operator	Precedence	Associativity	Arity	Type
`like`	1	non-assoc	dyadic	Relational
`<>`	1	left	dyadic	Relational
`>=`	1	left	dyadic	Relational
`<=`	1	left	dyadic	Relational
`=`	1	left	dyadic	Relational
`>`	1	left	dyadic	Relational
`<`	1	left	dyadic	Relational
`+`	2	left	dyadic	Arithmetic
`-`	2	left	dyadic	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

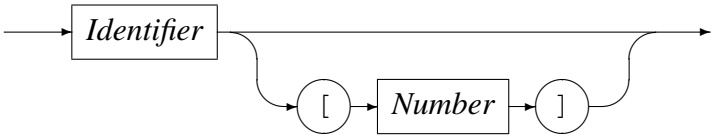
- Variables

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.

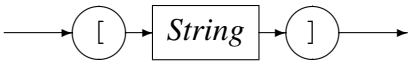
Variable



Variable Node



IndexedString



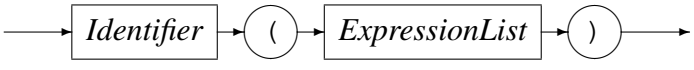
Examples of variable names are:

- `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

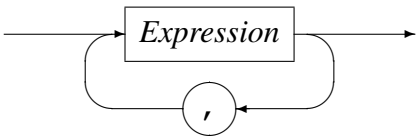
  A function is a special type of operator. It is specified by the function name, an identifier, followed by a comma separated list of arguments enclosed in parentheses.

  Function



where an 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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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`.

A.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.

A.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`.

A.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`.

A.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`.

A.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`.

A.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.

A.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).

A.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 A.3.11 on page 23

A.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.

A.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`.

A.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 5SWRel213144060219CommsType214INTERNAL MODEM 18TermType18EFTsma rt15OSVer198200360781 6SWHash18B4E1963A`

    returns the string `820036078`

A.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
    ```

A.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 is 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

A.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
    ```

A.3.20 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.

A.3.21 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.

A.3.22 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.

A.3.23 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.

A.3.24 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.

References

[1] expeval: Expression Evaluation User Guide. CML Document CML00091-01, Code Magus Limited, January 2013. PDF.

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