cmlmcase: Code Magus Cases Guide and Reference Version 3

CML00114-00

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

November 3, 2016

Contents

1 Introduction

Code Magus Cases allows artefacts bridging informal and formal entity descriptions to be used to select information and populate data sources used by other tools. Typically this data is to select entities for testing which satisfy specific conditions required by the test scenario.

Cases is often used within the Eresia Framework and fills the gap between content automation and test automation.

Although Cases is used with the Eresia Framework, it can be used with most testing tools that can represent their scenario data as text or as the contents of a spreadsheet, or as a part of the manual test effort. The only additional dependency on technology is the hosting operating system and its file system.

2 Command line parameters

```
Usage: cmlmcase [OPTION...]
  -i, --text-input=<template-file>                 Input template text file
  -r, --text-input-mode={r|<open-mode>}            Input template text file
                                                   open mode
  -o, --text-output=<edited-file>                  Output edited text file
  -w, --text-output-mode={w|<open-mode>}           Output edited text file
                                                   open mode
  -d, --data-file=<access>(<object>[,<options>])   Scenario candidate data
                                                   file: Recio open spec
  -t, --objtypes-name=<objtype-name>               Object types name used to
                                                   describe scenarios
  -s, --maximum-scenarios-per-type={1|<count>}     Maximum number of scenarios
                                                   to collect for each type
  -c, --charset={ascii|ebcdic}                     Default character set for
                                                   character data
  -e, --endian={big|little}                        Default binary data endian
  -b, --segment-tag={|<tag>}                       Name of segment to which
                                                   editing must be restricted
  -v, --verbose                                    Verbose output for problem
                                                   determination

Help options
  -?, --help                                       Show this help message
  --usage                                          Display brief usage message
```

where:

- `-i, --text-input=<template-file>`
  The input/template-file over which cases will be applied
- `-r, --text-input-mode={r|<open-mode>}`
  The mode of the text input file
- `-o, --text-output=<edited-file>`
  The output/edited-file containing the data supplied by cases
- `-w, --text-output-mode={w|<open-mode>}`
  The mode of the text output file
- `-d, --data-file=<access>(<object>[,<options>])`
- `-t, --objtypes-name=<objtype-name>`
  Name of objtypes member for buffer typing
- `-s, --maximum-scenarios-per-type={1|<count>}`
  Maximum scenarios to which cases will be applied per object type
- `-c, --charset={ascii|ebcdic}`
  Default character set for character data
- `-e, --endian={big|little}`
  Default binary data endian

- `-b, --segment-tag={|<tag>}`
  Tag of segment which should be edited
- `-v, --verbose`
  List symbol table and parsed copybooks and produce verbose run time messages during processing.

3 Processing

This program edits scenario data into a text file, which is very often a CSV file but for very wide cases the format should be the Code Magus export-excel format (or xxl) as produced by xlimpexp. These formats allow the text file after substitution to be reloaded into Excel and to be read by the corresponding Eresia portal; some other test tool or as input to a manual testing effort.

The input file, presented as text, is configured to select objects (records matching types) by placing certain tokens into the input file as placeholders for information from those objects.

As a consequence of running the tool over the input file, an updated text file will be written out in which elements belonging to particular scenario cases have been replaced.

The tokens available for use as place-holders are described in section 4 on page 6.

Additional input files include a file containing all the candidate case data as well as an object types member which defines and names all the scenario cases as defined by individual (sub)types.

A parameter specifies how many objects should match each of the given types. When the required number of types have been satisfied, or the end of the input file has been reached, the processing of the text input and output files begins.

A parameter indicates whether the text file to be edited contains tags to delimit editable segments of the text file. This parameter also supplies the value of the tag that must match the segment of the text file which will be edited. All other segments and the rest of the text file will be copied from the input to the output.

4 Tokens

Tokens defined within the Code Magus Cases tool act as placeholders for information matching certain scenarios/business rules/cases. This information, when matched to a particular scenarios/business rules/cases is substituted in the output file as a result of running the tool.

The following tokens are defined:

- `$$CASE-CONTEXT`
- `$$CASE-VALUE`
- `$$CASE-USABLE`
- `$$CASE-TITLE`
- `$$CASE-START-TAG`
- `$$CASE-END-TAG`
- `$$CASE-EXP`

Some tokens can be followed by a parameter in parenthesis, and for some tokens this is mandatory and sometimes it is optional.

- `$$CASE-CONTEXT`
  Establish an object matching a type as the current context. This token is followed by a mandatory parameter which must be the name of a type (or subtype). The next object matching that type is established as the current context, but the token and parameter are echoed back to the output text file.
- `$$CASE-VALUE`
  Insert a value from the current object into the output text: This token is used to select a a data item of the object currently in context and to replace the token with that data item's value in the output text. This token requires a parameter, the value of which is expected to be the fully qualified name of the data item. If there is currently no object in context or an error occurs during the processing of this token then the token and its parameter are copied to the output text.
- `$$CASE-USABLE`
  Test whether a case is usable or not. This token is used to indicate where the one the words "EXECUTABLE" or "INCOMPLETE" (in upper-case and without the quotation marks) should be placed in the output text. The token itself is not copied to the output text. The word "EXECUTABLE" is placed in the output text if there is a current object in context indicating that the last "$$CASES-CONTEXT", for example, found and established a matching object as the current context. The word "INCOMPLETE" is placed in the output text if there is no current object in context. This would be the case before the first occurrence of, for example, a

”$$CASE-CONTEXT” token, or as because the last attempt to establish a context failed to find a match.

The ”$$CASE-USABLE” can be followed by an optional parameter. This parameter is expected to be a type name. When this form is used an attempt it attempts to establish the next object of the named type as the current context and, depending on the success of this operation, replaces the token and parameter with the words ”EXECUTABLE” or ”INCOMPLETE” as described above. This option is a short-hand for a ”$$CASE-CONTEXT” token and parameter followed by a ”$$CASE-USABLE” token with only the effect of the latter being copied to the output text.

- `$$CASE-TITLE`
  Copy current context's title to output text. This token is used to replace, in the output text, the title of the current type in context. If no such object is in context then the token is copied to the output text without substitution.

- `$$CASE-EXP`
  This token allows replacement text to allow any expression over the matching buffers. Currently only one buffer is source for evaluating expressions and will be the last buffer introduced when defining multiple aliases or the current buffer matching the scenario when this is not the case. The expression evaluation syntax is explained fully in section 5.

5 Lexical Elements, Syntax and Semantics

5.1 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)`

5.2 Expression Grammar

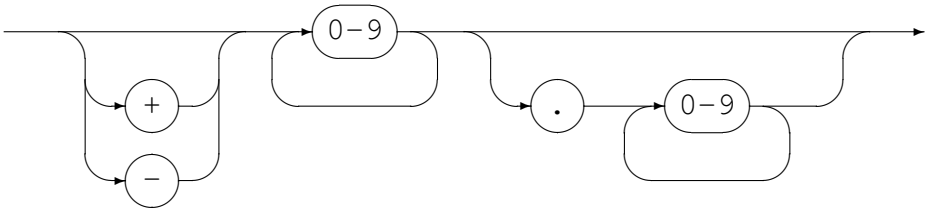
5.2.1 Lexical Elements

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

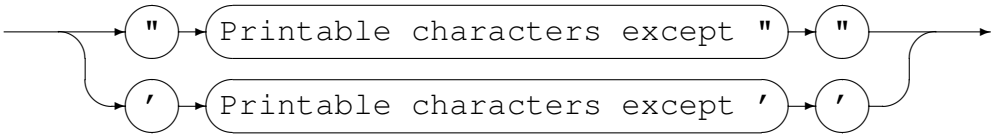


- String Literals

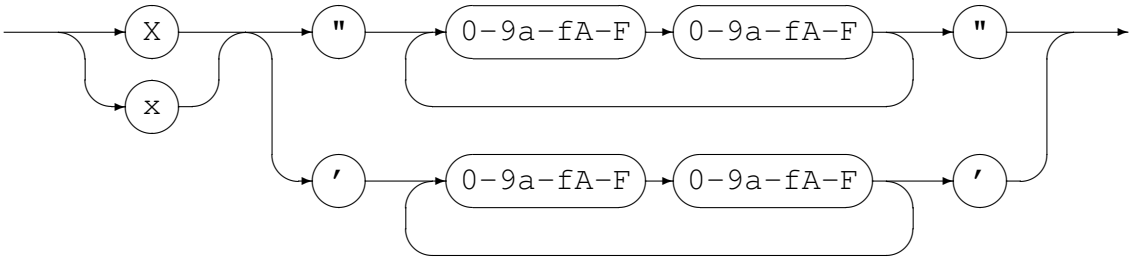
  String literals are made up from

  - 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

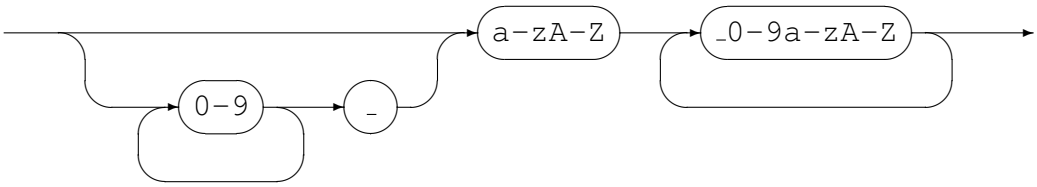


- Identifiers

  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

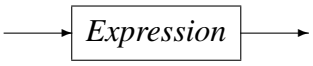


5.2.2 Syntactical Elements

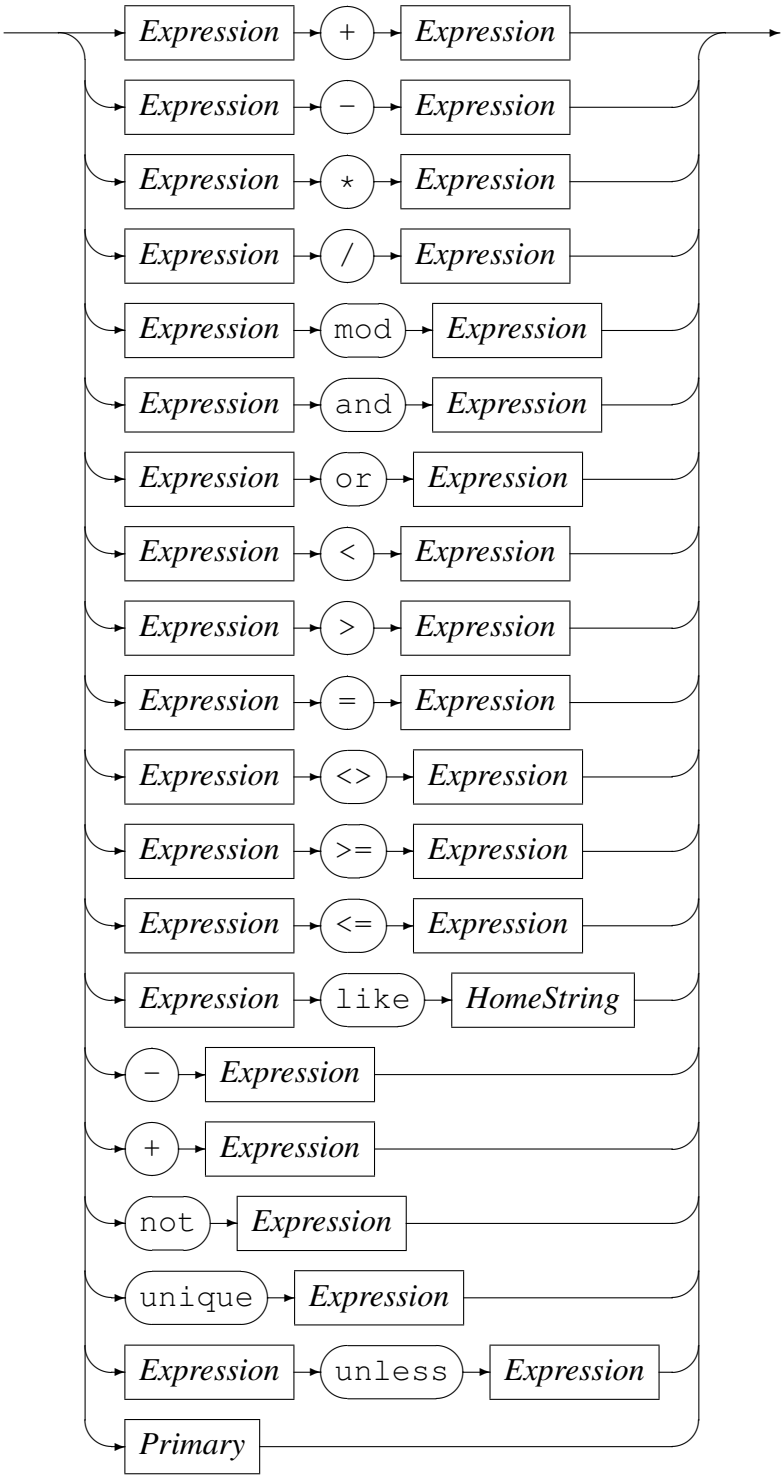
Expressions may themselves be used as syntactical elements when forming a compound expression.

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



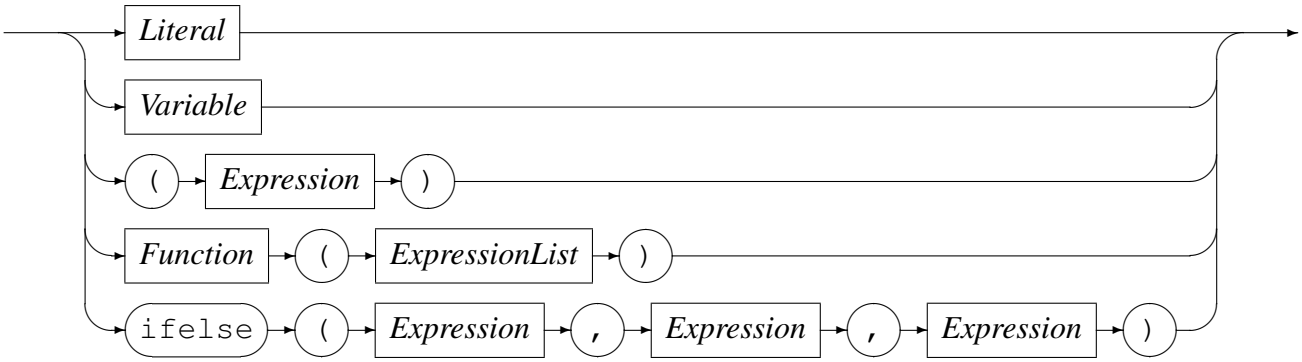
Expression



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 5.2.1 on page 8.

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 5.2.2 on page 14) 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 13 lists the allowed operators, their precedence, associativity, arity (whether or not they are monadic or dyadic) and Type.

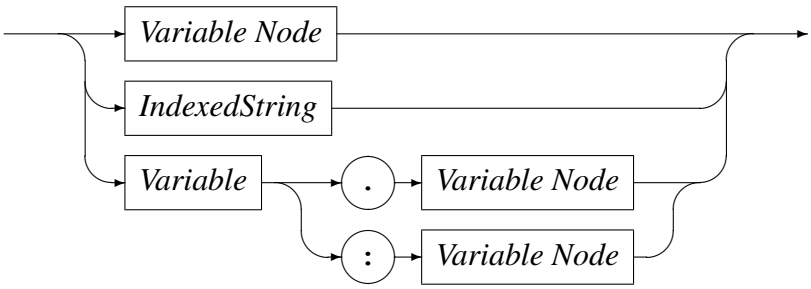
Operator	Precedence	Associativity	Arity	Type
<>	1	left	dyadic	Relational
>=	1	left	dyadic	Relational
<=	1	left	dyadic	Relational
=	1	left	dyadic	Relational
>	1	left	dyadic	Relational
<	1	left	dyadic	Relational
+	2	left	dyadic	Arithmetic
-	2	left	dyadic	Arithmetic
or	2	left	dyadic	Boolean
*	3	left	dyadic	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

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

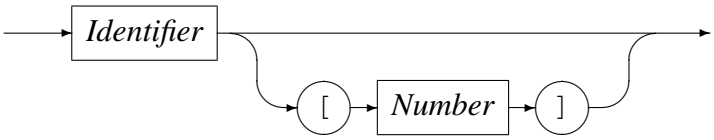
- Variables

  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.

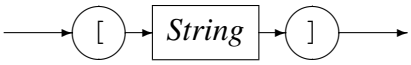
Variable



Variable Node



IndexedString



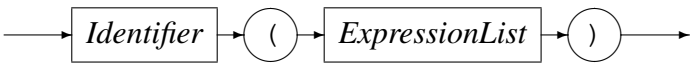
Examples of variable names are:

- `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

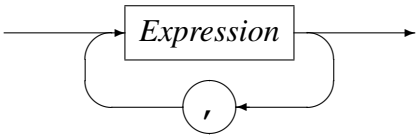
  A function is a special type of operator. It is specified by the function name, an identifier, followed by a comma separated list of arguments enclosed in parentheses.

  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.

5.3 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.

5.3.1 SysStrLen, strlen, length

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

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

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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`.

5.3.6 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.

5.3.7 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`.

5.3.8 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`.

5.3.9 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`.

5.3.10 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`.

5.3.11 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.

5.3.12 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).

5.3.13 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 5.3.11 on page 20

5.3.14 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 wether 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[1] 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[1] field `Record.Surname` occurs in the list of elements in the comma separated list specified by the first argument.

5.3.15 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`.

6 Examples

The example detailed below lists all the components required to apply the Code Magus Cases tool to a CSV file and generate the necessary output CSV file containing the substituted data.

The output CSV file would typically serve as input to an automated test cycle run by the Eresia Framework.

6.1 Copybook

The example copybook describes a set of fictional account data that could potentially apply to a wide range of industries.

```
01  ACCOUNTS.
    05 REC-TYPE              PIC X(4).
    05 ACCOUNT-NUM           PIC 9(8).
    05 ACCOUNT-STAT          PIC X(3).
    05 ACCOUNT-HOLDER        PIC X(20).
    05 STATUS-CODE           PIC X(3) OCCURS 3.
```

6.2 Objtypes

The types definition listed below is a very general view of the data described by the fictional account data. In other words the types file could potentially have contained a more in-depth view of the data and split the data into several types.

However, the aim of the example is make use of subtypes, as these translate well into business rules and are commonly used with the Code Magus Cases tool. They also provide a mechanism for extending existing types files without the need to update or duplicate existing objtypes artefacts.

To be clear the Code Magus Cases tool can make use of either objtypes or subtypes.

The choice will largely depend on existing metadata or how one chooses to separate out the business rules/cases.

For the purposes of the example subtypes will be the used but the parent type file from which the subtypes are derived is shown for the sake of completeness.

```
path ${HOME}"/testdata/copybooks/%s.cpy";
options ascii, endian_little, omit_fillers, extended_arith;

type TEST_ACCOUNTS
   title "Test ACCOUNT"
```

```
   book TESTACCT
   map ACCOUNTS
      include ACCOUNTS
;
```

6.3 Subtypes

The subtypes for the fictional account data may describe some business rule or data condition which is of interest. The subtypes used in the example are listed below.

```
path type ${HOME}"/testdata/objtypes/%s.objtypes";
path book ${HOME}"/testdata/copybooks/%s.cpy";

options ascii, endian_little, omit_fillers, extended_arith;

subtype AccAAA of cmlmcase_test_two:TEST_ACCOUNTS
   title "Account of type AAA"
   when (ACCOUNTS.ACCOUNT_STAT = "AAA");

subtype AccAAB of cmlmcase_test_two:TEST_ACCOUNTS
   title "Account of type AAB"
   when (ACCOUNTS.ACCOUNT_STAT = "AAB");

subtype AccAAC of cmlmcase_test_two:TEST_ACCOUNTS
   title "Account of type AAB"
   when (ACCOUNTS.ACCOUNT_STAT = "AAC");
```

6.4 Data

The fictional account data that will be applied to the example are listed below.

```
YYYY10129390AAAMRS JULIE ANN HARRIS777555111
XXXX10538423AAAMR IGNATIOUS PETERRS777555111
YYYY10832346AABDR RAYMOND A BARROSO777555111
YYYY10458567AAAPROF EDMUND CHAMBERS777555111
XXXX10438585AACLORD YURI GARGARISON777555111
YYYY10229884AAAMR DONALD DECIMALISE777555111
XXXX10997454AAAPROF ALBERT EINSTEIN777555111
XXXX12304832AABMRS MARGARET THATCHR777555111
YYYY14343536AAAMR EDWARD SCISSORHND777555111
YYYY10323636AACMR NICOLAS SARCOZIES777555111
YYYY10129903AAAMR DAVID C. CAMERSON777555111
XXXX10538234AAAMR BARRACUS OBAMMSON777555111
YYYY10832463AABMRS MARGERIN SIMPSON777555111
YYYY10458675AAAMR ELVIS GRACIA LEEU777555111
XXXX10438855AACMRS EDWINA EDWIN ELF777555111
YYYY10229848AAADR TIGERRS G WOODING777555111
XXXX10997544AAAMR ERNESTE RAMERSONG777555111
XXXX12304328AABMR LEOPOLDO GAMBLERS777555111
YYYY14343365AAAMRS MARCI PLATINO-LE777555111
YYYY10323366AACMR VICTOR MATTY FILD777555111
YYYY10193902AAAMRS MARTHA STEWARDES777555111
XXXX10584233AAAMRS NKOMBETHA NLAPPO777555111
YYYY10823463AABMR STEPHEN SPEELBERG777555111
YYYY10485675AAAMR DUEY DECIMALSYSTM777555111
XXXX10485853AACDR BEVERLY HILLS-LEE777555111
YYYY10298842AAAMRS LOVE-HEWITT ERIC777555111
XXXX10974549AAAMR PETER THE CHEATER777555111
XXXX12348320AABDR MONOPOLY THE GAME777555111
YYYY14335364AAAMR BARRONESS VON TRE777555111
YYYY10336362AACMR JONES ANDME SINGL777555111
YYYY10199032AAAMR AXL ROSIE-PARKERS777555111
XXXX10582343AAAMR STING ANTHEPOLICE777555111
YYYY10824633AABMR MIKE TYSON-PUNCHE777555111
```

6.5 CSV Input File

The input file to the Code Magus Cases example will be the CSV file listed below. The input file contains the tokens over which the account data will be substituted.

For the purposes of the example, a CSV file was chosen to demonstrate the concepts necessary to understand and implement cases, however the choice was arbitrary and as described in section 3, could also have been the Code Magus export-excel format (or xxl) as produced by xlimpexp or some other form of textual data depending on the user requirements.

```
TITLE OF PACKAGE,XYZAccountTester,,,
Test Cases (from row),6,,,
Test Cases (to row),15,,,
,,,,
STATUS OF SCENARIO, DESCRIPTION OF SCENARIO,CASE NAME,ACCOUNT_NUMBER,ACCOUNT_HOLDER
$$CASE-START(segment_one),,,,
$$CASE-CONTEXT(AccAAA)=$$CASE-USABLE;$$CASE-TITLE
$$CASE-USABLE(AccAAA),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-USABLE(AccAAC),$$CASE-NAME,$$CASE-EXP(SysSubStr(ACCOUNTS.ACCOUNT_HOLDER,1,10))END
$$CASE-USABLE(AccAAA),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-END,,,,
$$CASE-START(segment_two),,,,
$$CASE-CONTEXT(AccAAB)=$$CASE-USABLE;$$CASE-TITLE
$$CASE-USABLE(AccAAB),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-USABLE(AccAAC),$$CASE-NAME,$$CASE-EXP(SysSubStr(ACCOUNTS.ACCOUNT_HOLDER,1,10))END
$$CASE-USABLE(AccAAB),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-END,,,,
$$CASE-START(segment_three),,,,
$$CASE-CONTEXT(AccAAC)=$$CASE-USABLE;$$CASE-TITLE
$$CASE-USABLE(AccAAC),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-USABLE(AccAAC),$$CASE-NAME,$$CASE-VALUE(ACCOUNTS.ACCOUNT_NUM),$$CASE-VALUE(ACCOUNTS.STATUS_CODE[1])
$$CASE-USABLE(AccAAC),$$CASE-NAME,$$CASE-EXP(SysSubStr(ACCOUNTS.ACCOUNT_HOLDER,1,10))END
$$CASE-END,,,,
```

6.6 Command Line

The Code Magus Cases tool will be run on the command line and the account example was generated by running the command below. This command should be entered on one line to execute it; it is only formatted as below for readability.

```
cmlmcase -v -i input_files/CSVFILE_0003.csv
   -o output_files/CMLMCASE_OUT_0004.csv
   -d "text(input_files\DATAFILE_0008.txt,mode=r,texttype=UNIX)"
   -t testdata/objtypes/cmlmcase_test_two_subtypes.objtypes -s 10
```

6.7 The Output File

The output file will be generated as a result of running the tool with the required input file and command line options. The output file contains all the necessary data substitutions and would now be ready for use with the applicable testing tool.

```
TITLE OF PACKAGE,XYZAccountTester,,,
Test Cases (from row),6,,,
Test Cases (to row),15,,,
,,,,
STATUS OF SCENARIO, DESCRIPTION OF SCENARIO,CASE NAME,ACCOUNT_NUMBER,ACCOUNT_HOLDER
$$CASE-START(segment_one),,,,
$$CASE-CONTEXT(AccAAA)=EXECUTABLE;Account of type AAA//
EXECUTABLE,AccAAA,10538423,777
EXECUTABLE,AccAAC,LORD YURI
EXECUTABLE,AccAAA,10458567,777
$$CASE-END,,,,
$$CASE-START(segment_two),,,,
$$CASE-CONTEXT(AccAAB)=EXECUTABLE;Account of type AAB//
EXECUTABLE,AccAAB,12304832,777
EXECUTABLE,AccAAC,MR NICOLAS
EXECUTABLE,AccAAB,10832463,777
$$CASE-END,,,,
$$CASE-START(segment_three),,,,
$$CASE-CONTEXT(AccAAC)=EXECUTABLE;Account of type AAB//
EXECUTABLE,AccAAC,10323366,777
EXECUTABLE,AccAAC,10485853,777
EXECUTABLE,AccAAC,MR JONES A
$$CASE-END,,,,
```

References

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