# C.4 String/list handling functions

## C.4.1 The Regexp function

**regexp**(

**in** **template** **(value)** any\_character\_string\_type inpar,

**in** **template** **(present)** any\_character\_string\_type expression,

**in integer** groupno

) **return** any\_character\_string\_type

This function first matches the parameter inpar (or in case inpar is a template, its value equivalent) against the expression in the second parameter according to the pattern matching specified in clause B.1.5. If expression is not a template containing a pattern matching mechanism, it shall be processed by this predefined function as if it was a character pattern as described in clause B.1.5. If inpar is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents.

If this matching is unsuccessful, an empty string shall be returned.

If this matching is successful, the substring of inpar shall be returned, which matched the groupno-s group of expression during the matching. Group numbers are assigned by the order of occurrences of the opening bracket of a group and counted starting from 0 by step 1.

The parameters inpar and expression shall be a value or a template of **charstring** or **universal charstring** types. In case inpar is a template, it shall contain the specific value matching mechanism only. When expression is a template it shall contain the specific value or pattern matching mechanisms only. The parameter groupno shall be a non-negative integer. The type of the character string returned is the root type of inpar.

NOTE: This function differs from other well-known regular expression matching implementations in that:

a) It shall match the whole inpar string instead of only a substring.

b) It starts counting groups from 0, while in some other implementations the first group is referenced by 1 and the whole substring matched by the expression is referenced by 0.

In addition to the general error causes in clause 16.1.2, error causes are:

* when inpar is a template, it contains other matching mechanism than specific value or character pattern;
* when expression is a template, it contains other matching mechanism than specific value or character pattern;
* inpar is of charstring type and expression is of universal charstring type;
* groupno is a negative integer;
* there is no groupno -s group in expression.

EXAMPLE:

// Given

**var charstring** myInput := " simple text for a regexp example ";

**var charstring** myString;

myString := **regexp**(myInput,**charstring**:"?+(text)?+",0);

// will return "text"

myString := **regexp**(myInput,**charstring**:"?+(text)?+",1);

// causes an error as there is no group with index 1

myString := **regexp**(myInput,**charstring**:"(?+)(text)(?+)",0);

// will return " simple "

myString := **regexp**(myInput,**charstring**:"(?+)(text)(?+)",2);

// will return " for a regexp example "

myString := **regexp**(myInput,**charstring**:"((?+)(text)(?+))",0);

// will return the whole inpar, i.e. " simple text for a regexp example "

myString := **regexp**(myInput,**charstring**:"(([ ]+)(text)(?+))",0);

// will return an empty string as expression does not matches inpar

myString := **regexp**(myInput,**universal** **charstring**:"?+(text)?+",0);

// will cause an error as inpar is of type charstring, while

// expression is of type universal charstring

myInput := " date: 2001-10-20 ; msgno: 17; exp ";

**var** **template** **charstring** myPattern :=

**pattern** "([ \t]#(0,)date:[ \d\-]#(0,);[ \t]#(0,)msgno: (\d#(1,3)); (exp)#(0,1)) [ \t]#(0,)";  
 // please note, that only the very first opening bracket and the bracket before "\d#(1,3)"

// denotes groups; "#(0,)", "#(1,3)" and "#(0,1)" denotes matching the preceding expression

// several time

myString := **regexp**(myInput, myPattern,0);

// will return the input string but the whitespace at the end,

// i.e. " date: 2001-10-20 ; msgno: 17; exp"

myString := **regexp**(myInput, myPattern,1);

// will return the value "17"

//An example of a wrapper function to count groups from 1 and return the complete p\_inpar

//if p\_groupno equals 0

**function** regexp0(

**in** **template** **charstring** p\_inpar,

**in** **template** **charstring** p\_expression,

**in** **integer** p\_groupno)

**return** **charstring** {

**var** **template** **charstring** extended\_expr := **pattern** "({p expression})";

**return** **regexp**(p inpar, extended\_expr, p\_groupno )

}

## C.4.2 The Substring function

**substr**(  
 **in template** **(present)** any\_string\_or\_sequence\_type inpar,   
 **in integer** index,   
 **in integer** count  
) **return** input\_string\_or\_sequence\_type

This function returns a substring or subsequence from a value that is of a binary string type (**bitstring**, **hexstring**, **octetstring**), a character string type (**charstring**, **universal** **charstring**), or a sequence type (**record of**, **set of** or array). If inpar is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents. The type of the substring or subsequence returned is the root type of the input parameter. The starting point of substring or subsequence to return is defined by the second parameter (index). Indexing starts from zero. The third input parameter (count) defines the length of the substring or subsequence to be returned. The units of length for string types are as defined in table 4 of the present document. For sequence types, the unit of length is element.

NOTE: Please note that the root types of arrays is **record of**, therefore if inpar is an array the returned type is **record of**. This, in some cases, may lead to different indexing in inpar and in the returned value.

When used on templates of character string types, only the inside matching mechanisms *AnyElement* and *AnyElementsOrNone* are allowed in inpar and the function shall return the character representation of the matching mechanisms, i.e. "?" for *AnyElement* and "\*" for *AnyElementsOrNone*. When inpar is a template of binary string or sequence type or is an array, only the specific value and *AnyElement* matching mechanisms are allowed and the substring or subsequence to be returned shall not contain *AnyElement.*

In addition to the general error causes in clause 16.1.2, error causes are:

* index is less than zero;
* count is less than zero;
* index+count is greater than **lengthof**(inpar);
* inpar is a template of a character string type and contains a matching mechanism other than *AnyElement* or *AnyElementsOrNone*;
* inpar is a template of a binary string or sequence type or array and it contains other matching mechanism as specific value and *AnyElement*;
* inpar is a template of a binary string or sequence type or array and the substring or subsequence to be returned contains the *AnyElement* matching mechanism.

EXAMPLE:

**substr**('00100110'B, 3, 4) // returns '0011'B

**substr**('ABCDEF'H, 2, 3) // returns 'CDE'H

**substr**('01AB23CD'O, 1, 2) // returns 'AB23'O

**substr**("My name is JJ", 11, 2) // returns "JJ"

**substr**({ 4, 5, 6 }, 1, 2) // returns {5, 6}

## C.4.3 The Replace function

**replace**(  
 **in** any\_string\_or\_sequence\_type inpar,   
 **in integer** index,  
 **in integer** len,

**in** any\_string\_or\_sequence\_type repl  
) **return** any\_string\_or\_sequence type

This function replaces the substring or subsequence of value inpar at index index of length len with the string or sequence value repl and returns the resulting string or sequence. inpar shall not be modified. If len is 0 the string or sequence repl is inserted. If index is 0, repl is inserted at the beginning of inpar. If index is **lengthof**(inpar), repl is inserted at the end of inpar. If inpar is a literal (i.e. type is not explicitly given) the corresponding type shall be retrieved from the value contents. inpar and repl, and the returned string or sequence shall be of the same root type. The function replace can be applied to **bitstring**, **hexstring**, **octetstring**, or any character string, **record** **of**, **set** **of**, or arrays. Note that indexing in strings starts from zero.

NOTE: Please note that the root types of arrays is **record of**, therefore if inpar or repl or both are an array, the returned type is **record of**. This, in some cases, may lead to different indexing in inpar and/or repl and in the returned value.

In addition to the general error causes in clause 16.1.2, error causes are:

* inpar or repl are not of string, **record** **of**, **set** **of**, or array type;
* inpar and repl are of different root type;
* index is less than 0 or greater than **lengthof**(inpar);
* len is less than 0 or greater than **lengthof**(inpar);
* index+len is greater than **lengthof**(inpar).

EXAMPLE:

**replace** ('00000110'B, 1, 3, '111'B) // returns '01110110'B

**replace** ('ABCDEF'H, 0, 2, '123'H) // returns '123CDEF'H

**replace** ('01AB23CD'O, 2, 1, 'FF96'O) // returns '01ABFF96CD'O

**replace** ("My name is JJ", 11, 1, "xx") // returns "My name is xxJ"

**replace** ("My name is JJ", 11, 0, "xx") // returns "My name is xxJJ"

**replace** ("My name is JJ", 2, 2, "x") // returns "Myxame is JJ",

**replace** ("My name is JJ", 12, 2, "xx") // produces test case error

**replace** ("My name is JJ", 13, 2, "xx") // produces test case error

**replace** ("My name is JJ", 13, 0, "xx") // returns "My name is JJxx"