TECHNICAL REPORT

ISO/IEC TR 19075-1

First edition
2011-07-15

Information technology — Database languages — SQL Technical Reports —

Part 1: XQuery Regular Expression Support in SQL

Technologies de l'information — Langages de base de données — Rapport techniques SQL —

Partie 1: Support d'expressions régulières de XQuery en SQL

Reference number
ISO/IEC TR 19075-1:2011(E)

Contents

Page

(Blank page)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard (“state of the art”, for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 19075-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

ISO/IEC TR 19075 consists of the following parts, under the general title *Information technology — Database languages — SQL Technical Reports*:

— *Part 1: XQuery Regular Expression Support in SQL*

Introduction

The organization of this part of ISO/IEC TR 19075 is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC TR 19075.
2) Clause 2, “XQuery regular expressions”, explains how XQuery regular expressions are formed.
3) Clause 3, “Operators using regular expressions”, explains how the SQL operators use regular expressions.

Information technology — Database languages — SQL Technical Reports —

Part 1:
XQuery Regular Expression Support in SQL

1 Scope

This Technical Report describes the regular expression support in SQL adopted from the regular expression syntax of [XQuery F&O], which is derived from Perl. This Technical Report discusses five operators using this regular expression syntax:

— LIKE_REGEX predicate, to determine the existence of a match to a regular expression.

— OCCURRENCES_REGEX numeric function, to determine the number of matches to a regular expression.

— POSITION_REGEX function, to determine the position of a match.

— SUBSTRING_REGEX function, to extract a substring matching a regular expression.

— TRANSLATE_REGEX function, to perform replacements using a regular expression.

(Blank page)

2 XQuery regular expressions

XQuery regular expression syntax is specified in [XQuery F&O], section 7.6.1, “Regular expression syntax”. This paper references the XQuery specification, with two small modifications (required since character strings in an RDBMS are not necessarily normalized according to XML conventions). The following subsections provide an overview of this syntax.

The XQuery regular expression syntax is itself a modification of another regular expression syntax found in [XML Schema: Datatypes].

This section presents an overview of the capabilities of XQuery regular expression syntax. In the process, this section will illustrate some of the SQL operators. The SQL operators themselves are presented in the next section.

The following discussion does not cover every aspect of XQuery regular expressions; for this, [XQuery F&O] is the reference (though hardly a tutorial; a variety of popular works contain detailed treatments of regular expressions).

2.1 Matching a specific character

Perhaps the most elementary pattern matching requirement is the ability to match a single character or string. For most characters, this is done by simply writing the character in the regular expression. For example, suppose you want to know if a string S contains the letters “xyz”. This could be done with the following predicate:

```
S LIKE_REGEX 'xyz'
```

Note that the SQL LIKE predicate would require an exact match for “xyz”. However, the convention with regular expressions is that S need only contain a substring that is “xyz”. For example, all of the following values of S would yield *True* for the predicate above:

```
xyz
abcxyz123
1 xyz 2 xyz 3 xyz
```

Note that in the last example, there are actually three occurrences of the regular expression “xyz” within the tested value. The user may wish to know the number of occurrences of a match. This can be done with OCCURRENCES_REGEX. For example:

```
OCCURRENCES_REGEX ('xyz' IN '1 xyz 2 xyz 3 xyz') = 3
```

The user might also wish to know the position of a specific match. This can be done using POSITION_REGEX. For example, to learn the starting character position of the second occurrence,

```
POSITION_REGEX ( 'xyz' IN '1 xyz 2 xyz 3 xyz' OCCURRENCE 2 ) = 9
```

It is also possible to ask for the character position of the first character after the match. For example:

```
    POSITION_REGEX ( AFTER 'xyz' IN '1 xyz 2 xyz 3 xyz' OCCURRENCE 2 ) = 12
```

If AFTER is used and the last character of the subject string is consumed, then the result is the length of the string plus 1 (one):

```
    POSITION_REGEX ( AFTER 'xyz' IN 'xyz' ) = 4
```

2.2 Metacharacters and escape sequences

As mentioned, most characters can be matched by simply writing the character in the regular expression. However, certain characters are reserved as *metacharacters*. The complete list of metacharacters is:

```
    . \ ? * + { } ( ) | [ ] ^ $
```

The use of each of these metacharacters will be explained later. If you want to match a metacharacter, then you need to use an *escape sequence*, consisting of a backslash (“`/`”) followed by the metacharacter. For example, to test whether a string contains a dollar sign, you could write

```
    S LIKE_REGEX '\$'
```

In particular, the escape sequence representing a backslash is two consecutive backslashes. There are various other defined escape sequences, matching either a single character, or any of a group of characters. The *single character escape sequence*s are:

\n	newline (U+000A)
\r	return (U+000D)
\t	tab (U+0009)
\-	minus sign ('-')

The so-called *category escape*s are exemplified by “`\p{L}`” or “`\p{Lu}`”. A category escape begins with “`\p{`” followed by one uppercase letter, optionally a lowercase letter, and then the closing brace. In these example, “`\p{L}`” matches any letter (as defined by Unicode) and “`\p{Lu}`” matches any uppercase letter. Some interesting category escapes are listed below:

\p{L}	Any letter.
\p{Lu}	Any uppercase letter.
\p{Ll}	Any lowercase letter.
\p{Nd}	Any decimal digit.
\p{P}	Any punctuation mark.
\p{Z}	Any separator (space, line, paragraph, *etc*.).

The complete list of category escapes is found in [XML Schema: Datatypes], section F.1.1, “Character class escapes”.

There are also *complementary category escape*s, which are exemplified by “`\P{L}`” or “`\P{Lu}`”. A complementary category escape matches any character that would not be matched by the corresponding category escape. The difference is that the (positive) character escape is written with a lowercase “`p`” whereas the complementary character escape is written with an uppercase “`P`”.

The so-called *block escape*s match any character in a block of Unicode, that is, a predefined consecutive range of code points. For example, “`\p{IsBasicLatin}`” matches the ASCII character set. There are also *complementary block escape*s, such as “`\P{IsBasicLatin}`”, which matches any single character that is not an ASCII character.

Finally, there are the following *multi-character escape sequence*s:

\s	As defined by [XML Schema: Datatypes], this escape matches space (U+0020), tab (U+0009), newline (U+000A), or return (U+000D). Since character strings in an RDBMS have not undergone XML line termination normalization, we broaden it to include any character or two-character sequence that is recognized by [Unicode18] as a line terminator. Subclause 2.5, “Line terminators”, discusses this issue further.
\S	Any single character not matched by `\s`.
\i	Underscore (“`_`”), colon (“`:`”) or letter (this is a lot more than just the Latin letters; see [XML 1.0] appendix B, rule [84]).
\I	Any single character not matched by `\i`.
\c	Any single character matched by NameChar, as defined in [XML 1.0] section 2.3, rule [4].
\C	Any single character not matched by `\c`.
\d	Any single digit
\D	Any single character not matched by `\d`.
\w	Any single Unicode character except those classified as “punctuation”, “separator”, or “other”.
\W	The complement of `\w`.

2.3 Dot

Dot (period, “`.`”) is a metacharacter that is used to match any single character (the same behavior as “_” in LIKE predicates), or any single character that is not a line terminator. The default is to match anything except a line terminator. The alternative, called *dot-all mode*, is specified using a flag that contains a lowercase “`s`”.

For example

```
S LIKE_REGEX 'a.b'
```

matches the following:

```
'xa0by'
```

but not the following:

```
'xa
by'
```

because the character between the “`a`” and the “`b`” is a line terminator. However, using dot-all mode like this:

```
S LIKE_REGEX 'a.b' FLAG 's'
```

would match both examples.

2.4 Anchors

We have seen that regular expressions look for a match anywhere within a string, without needing to match the entire string. But what if you want to require a match of the entire string? For this, you can use *anchors*. The anchors are the metacharacters “`^`” for the start of a string (or line), and “`$`” for the end of a string (or line). For example:

```
S LIKE_REGEX '^xyz$'
```

can only match a string that is precisely 'xyz'.

Anchors may be used separately to require a “begins with” or “end with” match. For example

```
S LIKE_REGEX '^xyz'
```

matches any string that begins with “`xyz`”, and

```
S LIKE_REGEX 'xyz$'
```

matches any string that ends with “`xyz`”.

Instead of matching the begin or end of the string, the anchors may be used to anchor a match to the begin or end of a line, by performing the match in *multi-line mode*. Multi-line mode is specified using a flag containing a lowercase “`m`”. For example:

```
S LIKE_REGEX '^xyz' FLAG 'm'
```

performs an anchored search in multi-line mode, matching any string containing a line that begins with “`xyz`”. The example above would match the following string:

```
'line one
xyz
line three'
```

2.5 Line terminators

The metacharacters “`.`”, “`^`”, and “`$`” and the multi-character escape sequences “`\s`” and “`\S`” are defined in terms of a “line terminator”. What counts as a line terminator? [XQuery F&amp;O] only recognizes a line feed (U+000A) as a line terminator. This definition works well for XQuery, because XML normalizes the line terminators on various platforms to a line feed.

A closer look shows that XML has two definitions of line handling, in section 2.11, “End-of-line handling”, of [XML 1.0] and [XML 1.1]. So which should we use for SQL?

A first stop in answering this is to look at [SQL/XML WD] Subclause 6.17, “<XML query>”, which requires XML 1.0 as a basic level of support, and permits XML 1.1 support in the form of Feature X211, “XML 1.1

support". So, we might specify that the character string is normalized according to either XML 1.0 or XML 1.1 as an implementation-defined choice, or perhaps via a conformance feature.

However, some of the line terminators, even in XML 1.0, are two-character sequences. XML normalizes its input, which means that such two-character sequences are converted to a single character. This changes the relative position of every subsequent character, which would cause unexpected results for POSITION_REGEX.

Our solution is to look to [Unicode18], a Unicode standard containing guidelines for regular expression processors. This provides a referenceable definition of line terminator that does not require normalizing the subject character string.

2.6 Bracket expressions

So far, we have seen how to match a specific character, or any character from certain predefined sets of characters. Using bracket expressions, you can specify your own group of characters. (XML Schema and XQuery call these *character class expressions*, but the term *bracket expression* is in common use.)

A bracket expression is begun by a left bracket "`[`" and terminated by a right bracket "`]`". Bracket expressions have a different list of special characters, namely

```
^ [ ] \
```

For clarity, we will call these *special characters*, in contrast to the metacharacters listed earlier.

2.6.1 Listing characters

If a bracket expression does not contain any of the special characters, then the bracket expression matches any single character that is listed between the brackets. For example,

```
S LIKE_REGEX '[abc]'
```

matches any of the following:

```
'say'
'boy'
'lack'
```

All backslash escape sequences are available for use within a bracket expression. For example, to match either a caret or a backslash, you can use

```
S LIKE_REGEX '[\^\\]'
```

To match all letters or digits, one might use

```
S LIKE_REGEX '[\p{L}\p{Nd}]'
```

where "`\p{L}`" is the escape matching any letter and "`\p{Nd}`" is the escape matching any digit.

2.6.2 Matching a range

A minus sign "`-`" is used to specify a character range. For example:

```
S LIKE_REGEX '[sa-my]'
```

matches the lowercase letters "`s`", all the letters between "`a`" and "`m`" inclusive, and "`y`". Ranges are defined in terms of the UCS code point ordering. When there are multiple ranges, the bracket expression matches the union of the ranges. For example:

```
S LIKE_REGEX '[a-me-z]'
```

matches all lowercase letters.

Using a special character in a range is sometimes permitted, but tricky. Rather than present the rules here, our advice is to use a backslash escape if the start or end point of a range must be a special character.

2.6.3 Negation

A caret "`^`" is a special character when it is the first character of a bracket expression, where it indicates that the set of characters is anything not listed by the following bracket expression. For example:

```
S LIKE_REGEX '[^aj-m]'
```

is *True* if S contains any character that is not "`a`", "`j`", "`k`", "`l`", or "`m`".

2.6.4 Character class subtraction

A bracket expression may conclude with a minus sign "`-`" followed by a nested bracket expression. This is called a *character class subtraction*, and indicates that any character matched by the nested bracket expression is to be removed from the set of characters that might be a match. For example:

```
S LIKE_REGEX '[a-z-[m-p]]'
```

matches anything between "`a`" and "`z`", except for the letters between "`m`" and "`p`", inclusive. This example is equivalent to:

```
S LIKE_REGEX '[a-lq-z]'
```

Seemingly you can nest character class subtractions indefinitely. This concludes the presentation of bracket expressions.

2.7 Alternation

You can specify a choice of regular expressions using a vertical bar "`|`". For example:

```
S LIKE_REGEX 'a|b'
```

is *True* if S contains either an “a” or a “b”.

Alternation has lower precedence than concatenation. Thus

```
S LIKE_REGEX 'ab|xyz'
```

is *True* if S contains either “ab” or “xyz”. To override this precedence, you can use parentheses, such as this example:

```
S LIKE_REGEX 'a(b|xy)z'
```

The preceding example is *True* if S contains either “abz” or “axyz”.

2.8 Quantifiers

Quantifiers are metacharacters that specify a match for some number of repetitions of a regular expression. There are two sets of quantifiers, the greedy and the reluctant. The *greedy quantifier*s are:

{n}	Exactly n repetitions.
{n,}	*n* or more repetitions.
{n,m}	Between *n* and *m* repetitions, inclusive.
?	0 (zero) or 1 (one) repetition; equivalent to {0,1}.
*	0 (zero) or more repetitions; equivalent to {0,}.
+	1 (one) or more repetitions; equivalent to {1,}.

The *reluctant quantifier*s are formed by suffixing a question mark to a greedy quantifier. Thus, “*?” is the reluctant form of “*”, and “??” is the reluctant form of “?”. The greedy quantifiers try to match as much as possible, whereas the reluctant quantifiers try to match as little as possible (while still allowing the overall regular expression to match). There is no difference in behavior between the greedy and reluctant quantifiers for LIKE_REGEX. We will look at this difference for the other operators shortly.

Examples:

```
S LIKE_REGEX 'a{3}'
```

is equivalent to

```
S LIKE_REGEX 'aaa'
```

and matches any string containing at least three consecutive instances of “a”. Note that if S contains more than three consecutive instances of “a”, it still matches; to test whether S contains a substring of three consecutive instances of “a” and no more is a lot harder, since you have to also require something other than an “a” at both ends of the substring.

```
S LIKE_REGEX 'ab+c'
```

is equivalent to

```
S LIKE_REGEX 'ab{1,}c'
```

and matches any string that contains a substring consisting of an “a”, one or more “b”s, and then a “c”.

2.9 Locating a match

LIKE_REGEX only cares whether a match exists; the other operators care about where a match is located in the string. Consider the regular expression “a+” and the string “"a1aa2aaa3”. There are ten possible matches for “a+”, indicated by the underlining on the following lines:

```
'a1aa2aaa3' -- position 1, length 1

'a1aa2aaa3' -- position 3, length 1
'a1aa2aaa3' -- position 3, length 2
'a1aa2aaa3' -- position 4, length 1

'a1aa2aaa3' -- position 6, length 1
'a1aa2aaa3' -- position 6, length 2
'a1aa2aaa3' -- position 6, length 3
'a1aa2aaa3' -- position 7, length 1
'a1aa2aaa3' -- position 8, length 2
'a1aa2aaa3' -- position 9, length 1
```

Notice that some of the matches are substrings of other matches. The rules of XQuery regular expressions are designed to ignore certain matches, so that the recognized matches are mutually disjoint. Obviously there are many ways to do this, so the rules provide priorities in determining the recognized matches. There are three priorities:

1) The top priority is to find a match as early in the string as possible. This is commonly called the *leftmost rule*.

2) The second priority is to find the first alternative of an alternation, if possible. We are unaware of a common name for this rule.

3) The last priority is to find the longest possible match for greedy quantifiers, and the shortest match for reluctant quantifiers. In the case of greedy quantifiers, this is commonly called the *longest rule*; we are unaware of a common name for the rule regarding reluctant quantifiers.

[Historical note: POSIX only has a leftmost longest rule. There were no reluctant quantifiers, and the priority for matching alternations was the longest match rather than the first alternative.]

These rules will be illustrated by examples:

Subject string	**regular expression**	**match(es) underlined**	**priority**
baaaaaa	ba|a*	baaaaaa baaaaaa	leftmost (even though baaaaaa would be longer); second match must start after the first match
ab	a|ab	ab	first alternative (rather than matching ab)

Subject string	regular expression	match(es) underlined	priority
abcabbabc	ab*	abcabbabc abcabbabc abcabbabc	leftmost longest (greedy quantifier consumes two "b"s) longest
abcabbabc	ab*?	abcabbabc abcabbabc abcabbabc	shortest (no need to match "b") shortest shortest

2.10 Capture and back-reference

A *parenthesized sub-expression* is a portion of a regular expression that is enclosed in parentheses. Parenthesized sub-expressions are numbered in order of their left parenthesis. For example, in the regular expression

```
((a)|(b))
```

there are three sub-expressions:

1) `((a)|(b))`

2) `(a)`

3) `(b)`

A sub-expression can be referenced later in a regular expression using a back-reference, taking the form of a backslash followed by one or more digits. Thus the three sub-expressions in the example can be referenced as “`\1`”, “`\2`”, and “`\3`”. For example, consider the regular expression:

```
\p{Z}(\p{L}*)\p{Z}*\1\p{Z}
```

The first and only parenthesized sub-expression (“`\p{L}*`”) matches any sequence of letters that is bounded by some kind of space character (“`\p{Z}`”) before and after the sequence of letters. The back-reference (“`\1`”) matches whatever sequence of letters was captured by the first sub-expression. This regular expression might be used to search for occurrences of a repeated word (perhaps caused by a cut-and-paste error). Here is an example of a subject string, with underlining to indicate the match for the entire regular expression:

```
Hello Dolly you're looking looking swell
```

When a back-reference references a parenthesized group with a quantifier, then the back-reference matches the last iteration of the quantified sub-expression. For example, consider the regular expression:

```
'(ab*)*c*\1'
```

and the subject string:

```
'abbbabbabcabbbbb'
```

The matches to “`(ab*)`” are shown by underlining below:

```
'abbbabbabcabbbbb'
```

```
'abbbabbabcabbbbb'
'abbbabbabcabbbbb'
```

These three iterations of “`(ab*)`” are matched by “`(ab*)*`” and then the “`c`” is matched. Next, we need to match “`\1`”. The last match for the first parenthesized sub-expression is “`ab`”, so the overall match is indicate by underlining below:

```
'abbbabbabcabbbbb'
```

In the event that a sub-expression is unmatched, a back-reference to it matches the zero-length string. For example, consider the regular expression:

```
'((a*)|(b*))c??\3'
```

and the subject string:

```
'xyzaaccb'
```

In this example, the alternation “`((a*)|(b*))`” matches the “`aa`”, which is a match for the first alternative. Thus there is no match for the second alternative, “`(b*)`”. The “`c?`” prefers to match a zero-length string (though it could match the “`c`”), and the “`\3`” must match a zero-length string. Thus, the complete substring that is matched is underlined below:

```
'xyzaaccb'
```

2.11 Precedence

The precedence of operators outside bracket expressions is as follows (from highest to lowest):

— Highest precedence: atoms, defined as:

- Parentheses.
- Individual characters.
- Escape sequences.
- Dot (“`.`”)
- Anchors (“`^`”, “`$`”)
- Bracket expressions.

— Quantifiers.

— Concatenation.

— Alternation (lowest).

Examples:

1) Quantifiers have higher precedence than concatenation:

   `ab*` is equivalent to `a(b*)`

2) Concatenation outranks alternation:

   `ab|cd` is equivalent to `(ab)|(cd)`

2.12 Modes

The preceding discussion has mentioned two of the flags, “`s`” to specify dot-all mode, and “`m`” to specify multi-line mode. There are two additional flags, “`i`” for case-insentive mode, and “`x`” to disregard white space in regular expressions for readability. The complete set of modes is:

"s" Specifies dot-all mode, in which a period matches any character. If “`s`” is not specified, then a period matches any single character except a line terminator.

"m" Specifies multi-line mode, in which the anchors match the beginning or end of a line. If “`m`” is not specified, then the anchors match the beginning or end of the subject string.

"i" Specifies case-insensitive mode.

"x" Specifies that white space characters in a regular expression are ignored. This allows you to set off portions of a regular expression for greater readability.

(Blank page)

3 Operators using regular expressions

SQL contains five operators that use the XQuery regular expression syntax:

1) LIKE_REGEX — predicate that returns *True* if a substring of a string matches a regular expression.
2) OCCURRENCES_REGEX — numeric function returning the number of matches for a regular expression in a string.
3) POSITION_REGEX — numeric function returning the position of the start of a match for a regular expression in a string, or the position of the next character after a match.
4) SUBSTRING_REGEX — character string function returning a substring that matches a regular expression in a string.
5) TRANSLATE_REGEX — character function that performs a replacement operation on one or all matches to a regular expression in a string.

3.1 LIKE_REGEX

LIKE_REGEX is a predicate that returns *True* if a substring of a string matches a regular expression.

The syntax is:

```
<regex like predicate> ::=
    <row value predicand>
        [ NOT ] LIKE_REGEX <XQuery pattern>
        [ FLAG <XQuery option flag> ]
```

where

— <row value predicand> is the subject string to be searched for matches to the <XQuery pattern>.

— <XQuery pattern> is a character string expression whose value is an XQuery regular expression.

— <XQuery option flag> is an optional character string, corresponding to the $flags argument of the [XQuery F&O] function `fn:match`.

The result is *Unknown* if any of the operands is the null value, *True* if there is a substring that matches the <XQuery pattern> in the <row value predicand>, and *False* if there is no match.

Note that unlike LIKE, LIKE_REGEX can return *True* without matching the entire string. The usual convention for regular expression matching is to search for a match somewhere within the searched string, without necessarily matching the entire string. The user may use anchors to require a match to the entire string.

Exceptional cases:

— If any of the parameters is the null value, the result is *Unknown*.

— If the pattern or flag is not valid, then an exception condition is raised.

Examples:

`'abcde' LIKE_REGEX 'c'` evaluates to *True*.

`'abcde' LIKE_REGEX 'x'` evaluates to *False*.

`'abcde' LIKE_REGEX CAST (NULL AS CHAR(10))` evaluates to *Unknown*.

`'abcde' LIKE_REGEX '\'` raises an exception condition. In this example, “\” is not a well-formed regular expression.

`'abcde' LIKE_REGEX 'x' FLAG '?'` raises an exception condition. In this example, the flag “?” is invalid.

3.2 OCCURRENCES_REGEX

OCCURRENCES_REGEX is a numeric function returning the number of matches for a regular expression in a string. The syntax is:

```
<regex occurrences function> ::=
    OCCURRENCES_REGEX <left paren>
        <XQuery pattern> [ FLAG <XQuery option flag> ]
        IN <regex subject string>
        [ FROM <start position> ]
        [ USING <char length units> ] <right paren>
```

where:

— <XQuery pattern> is a character string expression whose value is an XQuery regular expression.

— <XQuery option flag> is an optional character string, corresponding to the $flags argument of the [XQuery F&O] function fn:match.

— <regex subject string> is the character string to be searched for matches to the <XQuery pattern>.

— <start position> is an optional exact numeric value with scale 0 (zero) specifying the position at which to start the search (the default is position 1 (one)).

— <char length units> is CHARACTERS or OCTETS, indicating the unit in which <start position> is measured (the default is to measure in CHARACTERS).

The <regex subject string> is searched for matches to the <XQuery pattern>, starting from position <start position>, which is measured in the units specified by <char length units>, either CHARACTERS or OCTETS. The result is the number of matches.

Exceptional cases:

— If any of the parameters is the null value, then the result is the null value.

— If the pattern or flag is not valid, then an exception condition is raised.

— If a starting position is given in octets, but it is not the first octet of a character, then the result is implementation-dependent. The use of OCTETS is discussed under POSITION_REGEX.

— If any of the numeric parameters is too large or too small, then the result is –1. This includes the following cases:

- The starting position is less 1 (one).
- The starting position is greater than the length of the string (measured in CHARACTERS or OCTETS as specified by <char length units>).

Examples:

`OCCURRENCES_REGEX ( 'a' IN 'what is that?' )` evaluates to 2.

`OCCURRENCES_REGEX ( 'a' IN 'what is that?' FROM 5)` evaluates to 1 (one).

`OCCURRENCES_REGEX ( 'A' FLAG 'i' IN 'what is that' )` evaluates to 2.

`OCCURRENCES_REGEX ( 'A' IN 'what is that' )` evaluates to 0 (zero).

3.3 POSITION_REGEX

POSITION_REGEX is a numeric function returning the position of the start of a match, or one plus the end of a match, for a regular expression in a string. The syntax is:

```
<regex position expression> ::=
    POSITION_REGEX <left paren> [ <regex position start or after> ]
        <XQuery pattern> [ FLAG <XQuery option flag> ]
        IN <regex subject string>
        [ FROM <start position> ]
        [ USING <char length units> ]
        [ OCCURRENCE <regex occurrence> ]
        [ GROUP <regex capture group> ] <right paren>

<regex position start or after> ::=
    START
  | AFTER
```

where:

— START indicates that the starting position of the match to the regular expression is desired; AFTER indicates that the character position immediately following the match is desired (START is the default). If the match consumes the last character of the subject string, then AFTER returns the length of the string plus 1 (one).

— <XQuery pattern> is a character string expression whose value is an XQuery regular expression.

— <XQuery option flag> is an optional character string, corresponding to the `$flags` argument of the [XQuery F&O] function `fn:match`.

— <regex subject string> is the character string to be searched for matches to the <XQuery pattern>.

— <start position> is an optional exact numeric value with scale 0 (zero), identifying the character position at which to start the search (the default is 1 (one)).

— <char length units> is CHARACTERS or OCTETS, indicating the unit in which <start position> is measured, and the unit in which the returned position is measured (the default is to measure in CHARACTERS).

— <regex occurrence> is an optional exact numeric value with scale 0 (zero) indicating which occurrence of a match is desired (the default is 1 (one)).

— <regex capture group> is an optional exact numeric value with scale 0 (zero) indicating which capture group of a match is desired (the default is 0 (zero), indicating the entire occurrence).

The <regex subject string> is searched for matches to the <XQuery pattern>. If there are at least *RO* matches, where *RO* is the value of <regex occurrence>, then either the starting position of the *RO*-th match, or the position immediately following the *RO*-th match, is returned (for the START or AFTER options, respectively). Positions are measured in the units specified by <char length units>, either CHARACTERS or OCTETS. If a <regex capture group> *CAP* is specified, then the position at the start or immediately following the substring that matches the *CAP*-th parenthesized subexpression is used.

With AFTER, note that the position returned is the one after the match. If the match consumes the last character of the string, then the position returned is actually one plus the length of the string (in characters or octets, as requested by <char length units>). The rationale for providing the position that is 1 (one) after the end of the match is that this is the correct place to begin a search for the next match. If the user wishes to process the subject string in a loop, the loop can continue until the AFTER position is greater than the length of the subject string. However, when doing this, the user must beware of a pitfall: if the regular expression matches a zero-length string, then the AFTER position and the START position are the same, and resuming the search at the AFTER position will simply find the same zero-length match again.

OCTETS is provided for efficient processing for those UCS encodings that do not have a fixed character width. It is expected that the user will use the output of POSITION_REGEX (... USING OCTETS ...) to learn the position of some occurrence within a string, measured in octets. That value is then known to be the first octet of a character, and may be used as a starting position in other function invocations. If the user picks an arbitrary octet number, it may be other than the first octet of a character. Naturally, beginning a regular expression match at such an octet can produce unpredictable results. Therefore we say that the result is implementation-dependent if a starting octet is not the first octet of a character.

Exceptional cases:

— If any of the parameters is the null value, the result is the null value.

— If the pattern or flag is not valid, then an exception condition is raised.

— If a starting position is given in octets, but it is not the first octet of a character, then the result is implementation-dependent.

— If any of the numeric parameters is too large or too small, then the result is 0 (zero). This includes the following cases:

- The starting position is less 1 (one).
- The starting position is greater than the length of the string (measured in CHARACTERS or OCTETS as specified by <char length units>).
- There are not at least *RO* matches.
- There are not *CAP* parenthesized subexpressions.

Examples:

`POSITION_REGEX ( 'a' IN 'what is that?' )` evaluates to 3.

`POSITION_REGEX ( START 'a' IN 'what is that?' )` evaluates to 3.

`POSITION_REGEX ( AFTER 'a' IN 'what is that?' )` evaluates to 4.

`POSITION_REGEX ( AFTER 'a' IN 'a')` evaluates to 2.

`POSITION_REGEX ( 'a' IN 'what is that?' FROM 5)` evaluates to 11.

`POSITION_REGEX ( 'a' IN 'what is that?' OCCURRENCE 2 )` evaluates to 11.

`POSITION_REGEX ( '(a)(t)' IN 'what is that?' GROUP 2 )` evaluates to 4.

`POSITION_REGEX ( 'A' FLAG 'i' IN 'what is that' )` evaluates to 3.

`POSITION_REGEX ( 'A' IN 'what is that' )` evaluates to 0.

3.4 SUBSTRING_REGEX

SUBSTRING_REGEX is a character string function returning a substring that matches a regular expression in a string. The syntax is:

```
<regex substring function> ::=
    SUBSTRING_REGEX <left paren>
        <XQuery pattern> [ FLAG <XQuery option flag> ]
        IN <regex subject string>
        [ FROM <start position> ]
        [ USING <char length units> ]
        [ OCCURRENCE <regex occurrence> ]
        [ GROUP <regex capture group> ] <right paren>
```

where:

— <XQuery pattern> is a character string expression whose value is an XQuery regular expression.

— <XQuery option flag> is an optional character string, corresponding to the `$flags` argument of the [XQuery F&O] function `fn:match`.

— <regex subject string> is the character string to be searched for matches to the <XQuery pattern>.

— <start position> is an optional exact numeric value with scale 0 (zero), indicating the character position at which to start the search (the default is position 1 (one)).

— <char length units> is CHARACTERS or OCTETS, indicating the unit in which <start position> is measured (the default is to measure in CHARACTERS).

— <regex occurrence> is an optional exact numeric value with scale 0 (zero) indicating which occurrence of a match is desired (the default is 1 (one)).

— <regex capture group> is an optional exact numeric value with scale 0 (zero) indicating which capture group of a match is desired (the default is 0 (zero), indicating the entire occurrence).

The <regex subject string> is searched for matches to the <XQuery pattern>. If there are at least *RO* matches, where *RO* is the value of <regex occurrence>, then the result is the substring that is the *RO*-th match. If <regex capture group> *CAP* is specified, then the result is the substring that matches the *CAP*-th parenthesized substring within the substring that is the *RO*-th match. If there are not at least *RO* matches, or at least *CAP* parenthesized subexpressions, the result is the null value.

The exceptional cases are:

— If any of the parameters is the null value, the result is the null value.

— If the pattern or flag is not valid, then an exception condition is raised.

— If a starting position is given in octets, but it is not the first octet of a character, then the result is implementation-dependent.

— If any of the numeric parameters is too large or too small, then the result is the null value. This includes the following cases:

- The starting position is less than 1 (one).
- The starting position is greater than the length of the string (measured in CHARACTERS or OCTETS as specified by <char length units>).
- There are not at least *RO* matches.
- There are not *CAP* parenthesized subexpressions.

Examples:

`SUBSTRING_REGEX ( '\p{L}*' IN 'what is that?' )` evaluates to the string “`what`”.

`SUBSTRING_REGEX ( '\p{L}*' IN 'what is that?' FROM 2)` evaluates to the string “`hat`”.

`SUBSTRING_REGEX ( '\p{L}*' IN 'what is that?' OCCURRENCE 2)` evaluates to the string “`is`”.

`SUBSTRING_REGEX ('(is) (\p{L}*)' IN 'what is that?' GROUP 2 )` evaluates to the string “`that`”.

3.5 TRANSLATE_REGEX

TRANSLATE_REGEX is a character string function that performs a replacement operation on one or all matches to a regular expression in a string. The syntax is:

```
<regex transliteration> ::=
    TRANSLATE_REGEX <left paren>
        <XQuery pattern> [ FLAG <XQuery option flag> ]
        IN <regex subject string>
        [ WITH <regex replacement string> ]
        [ FROM <start position> ]
        [ USING <char length units> ]
        [ OCCURRENCE <regex transliteration occurrence> ] <right paren>

<regex transliteration occurrence> ::=
    <regex occurrence>
  | ALL
```

where:

— <XQuery pattern> is a character string expression whose value is an XQuery regular expression.

— <XQuery option flag> is an optional character string, corresponding to the `$flags` argument of the [XQuery F&O] function `fn:match`.

— <regex subject string> is the character string to be searched for matches to the <XQuery pattern>.

— <regex replacement string> is a character string whose value is suitable for use as the `$replacement` argument of the [XQuery F&O] function `fn:replace`. The special syntax for replacement strings is discussed below. The default is the zero-length string.

— <start position> is an optional exact numeric value with scale 0 (zero), indicating the character position at which to start the search (the default position is 1 (one)).

— <char length units> is CHARACTERS or OCTETS, indicating the unit in which <start position> is measured (the default is to measure in CHARACTERS).

— <regex transliteration occurrence> is either the keyword ALL, or an exact numeric value with scale 0 (zero), indicating which occurrence of a match is desired (the default is ALL).

The <regex subject string> is searched for matches to the <XQuery pattern>. If ALL is specified or implied, then every match is replaced by the value of <regex replacement string>. If a numeric <regex transliteration occurrence> is specified, then only that match is replaced.

Exceptional cases:

— If any of the parameters is the null value, then the result is the null value.

— If the pattern, flag or replacement string is not valid, then an exception condition is raised.

— If the pattern matches a zero-length string, then an exception condition is raised (this is the behavior of XQuery's `fn:replace` in this case)

— If a starting position is given in octets, but it is not the first octet of a character, then the result is implementation-dependent.

— If any of the numeric parameters is too large or too small, then the result is the null value. This includes the following cases:

- The starting position is less than 1 (one).
- The starting position is greater than the length of the string (measured in CHARACTERS or OCTETS as specified by <char length units>).
- A numeric <regex transliteration occurrence> is specified, and there are not at least that many matches.

First, here are some examples with no replacement string. In these examples, any matched substring is replaced by a zero-length string, effectively removing the matched substring.

`TRANSLATE_REGEX ('a' IN 'what was that?')` evaluates to the string “wht ws tht?”

`TRANSLATE_REGEX ('a' IN 'what was that?' OCCURRENCE ALL)` evaluates to the string “wht ws tht?”

`TRANSLATE_REGEX ('a' IN 'what was that?'FROM 5)` evaluates to the string “what ws tht?”

`TRANSLATE_REGEX ('a' IN 'what was that?' OCCURRENCE 2)` evaluates to the string “what ws that?”

`TRANSLATE_REGEX ('A' IN 'what was that?')` evaluates to the string “what was that?”

`TRANSLATE_REGEX ('A' FLAG 'i' IN 'what was that?' )` evaluates to the string “wht ws tht?”

Next, here are some examples in which the matched substrings are replaced with constant text: