Regex Tutorial v02

Given an arbitrary string of characters, a regular expression can be used to determine if the string of characters matches a pattern. There is no standard for the definition of a regular expression, but two of the most common (and of most interest to the YANG community) are PCRE2 (Perl-compatible regular expressions) and the RE Library in Python.

For basic information, there are two useful websites. Regular-Expressions.info ( <https://www.regular-expressions.info/quickstart.html>) provides a quick start with example and terminology that will help the reader. The quick reference (<https://www.regular-expressions.info/refquick.html>) is also useful when trying to understand what a regular expression means. A tool that is very valuable is regex101(<https://regex101.com/>) which has various flavors of regular expressions, and a way to enter your own regular expressions and test input and get immediate feedback. The YANG Catalog also has a regex expression validator (<https://www.yangcatalog.org/yangre>) that can be used to check YANG or W3C validation. The yangre tool also as a github site (<https://github.com/YangCatalog/yangre>).

Regular expressions are very powerful and can be used for match string and/or string substitution. This appendix is only looking at using regular expressions in YANG pattern statements to validate string content.

Starting with a simple example:

leaf org-name {

type string {

pattern 'IEEE.\*';

}

}

In the above YANG there is a string leaf called “org-name” that has a pattern that will match any string that includes the letters IEEE followed by any number of other characters.

So given a json snippet that looks like this:

"org-name": "IEEE802"

The match is successful.

However if the input looked like this:

"org-name": "IEE802"

Then the match will fail because there are is not an “I” followed by three “E”s.

Regular expressions get interesting when the desire is to match a range of characters and there are many capabilities to explore. The most pertinent capabilities to understand to read the patterns in AXdz are:

## Range Match:

For example, the goal is match only a single digit between 0 and 5 inclusive. So 3 is valid but 7 is not.

leaf range-test-00 {

type string {

pattern '[0-5]';

}

The above pattern in the leaf range-test says that:

[0-5] means any number between 0 and 5 inclusive.

## Range Match with repetition:

For example, the goal is match only numbers that are contain 1, 2, or 3 digits and that do not contain any single digit that is greater than 5. So 123 is valid but 126 is not, or 3 is valid and 7 is not.

leaf range-test-01 {

type string {

pattern '[0-5]{1,3}';

}

The above pattern in the leaf range-test says that:

[0-5] means any number between 0 and 5 inclusive.

{1,3} means between 1 and 3 repetitions of the pattern.

## Grouping:

For example, the goal is to match one of two or more patterns). In the example below, it is desired to match either a single digit number between 0-9 inclusive or a two-digit number between 10 and 99 inclusive. So 0 is valid but 00 is not.

leaf range-test-02 {

type string {

pattern '([0-9]{1}|[1-9]{1}[0-9]{1})';

}

The yellow part says one instance {1} of any number between 0 and 9 inclusive. The green part (the vertical bar) is an “or”. The blue and purple parts make up a two digit number that will be between 10 and 99 inclusive. Not the use of the {1} is not strictly necessary because if you leave that out, one instance is assumed. It is good practice to include it so that it is very clear what is going on.

## Repetition – Star and Question mark:

Two other repetition options that are useful include the star (\*) which indicates the pattern before it can happen 0 to any number of times (sometimes called a quantifier). The question mark (?) in certain contexts is a repetition operator that means the pattern before it can happen 0 or 1 time only.

The first example is to have a single digit number between 0 to 9 inclusive, followed by a dash, followed by any number of single digits.

leaf range-test-03 {

type string {

pattern '([0-9]{1}-[0-9]\*)';

}

In the pattern above the strings “0-“ and “0-111111”, but “0-A” and “-1” would not. The yellow part matches a single digit between 0 and 9 inclusive. Followed by a dash, followed by the blue part which says any amount of digits between 0 and 9 inclusive.

If you replace the star (\*) with a question mark (?) in the above pattern, that would mean either no digits after the dash or a single digit after the dash. “0-“, and “0-0” would match but “0-11” would not.

The above syntax is all that is needed to understand the following pattern.

## The AXdz range pattern

The desire is to be able to specify masks in a compact way. There are 4096 bits, so the ability to state that you want a mask that supports individual numbers and ranges separated by commas.

So for example valid inputs would be:

* 1,5,6-10,2000-3000,4094
* 0
* 1,2,3,4,4094

Invalid input would be anything that falls outside of 0 to 4095 inclusive. There are error cases that are difficult to catch using regular expressions, for example ranges that go from a larger number to a smaller number, but that could be caught in post processing.

The pattern used is:

'([0-9]{1}|[1-9]{1}[0-9]{1,2}|[1-3]{1}[0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1})' +

'(-([1-9]{1}[0-9]{0,2}|[1-3][0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1}))?' +

'(,([1-9]{1}|[1-9]{1}[0-9]{1,2}|[1-3]{1}[0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1})' +

'(-([1-9]{1}[0-9]{0,2}|[1-3][0-9]{3}|40{1}[0-8]{1}[0-9]{1}|409[0-5]{1}))?)\*’

Looks daunting, but breaking it down into its parts helps…

### Part 1

([0-9]{1}|[1-9]{1}[0-9]{1,2}|[1-3]{1}[0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1})

This has options for a single digit number between 0 to 9 inclusive, a two or three digit number between 100 to 999 inclusive, a four digit number from 1000-3999 inclusive, and finally a four digit number between 4000 and 4095 inclusive. This pattern matches any number between 0 and 4095 inclusive.

### Part 2

(-([1-9]{1}[0-9]{0,2}|[1-3][0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1}))?

The important element here is the question mark (?) at the end, because that says there will be 0 or 1 of the pattern. So in the case of something like 1000-2000, that would match. If there was just a single digit, that would match part 1 and ignore part 2 because part 2 is optional.

### Part 3 and Part 4

(,([1-9]{1}|[1-9]{1}[0-9]{1,2}|[1-3]{1}[0-9]{3}|40[0-8]{1}[0-9]{1}|409[0-5]{1})(-([1-9]{1}[0-9]{0,2}|[1-3][0-9]{3}|40{1}[0-8]{1}[0-9]{1}|409[0-5]{1}))?)\*

Part 3 and Part 4 are just a copy of Part 1 and Part 2 with two additions. The yellow part looks for a comma. So this is what makes a string like 1,2 valid. Because the 1 would be matched my Part 1 and the ,2 would be matched by Part 3. The star (\*) at the end (shown in green) indicates that Part 3 and Part 4 can occur 0 to any number of times.

An example like: “1,10-22,4095” matches the above pattern. The “1” is matched by Part 1 and Part 2 is ignored because there is no dash. So “,10” is matched by Part 3 and “-22” is matched by Part 4. The “,4095” is matched by Part 3 and Part 4 is ignored because there is no dash (and it is optional because of the question mark (?) shown in blue.