Functions

Functions are used to perform specific manipulations on parameters in templates. The generic format is `[function_name(function_arguments)]`. Functions are part of the template line or its condition and can be mixed with conditionals and other parameter contexts.

Calculation functions

Calculation functions are meant to manipulate parameter values for use in templates. Hereby the database values remain unchanged. The calculation functions are used in template lines for substitution and conditional purposes. Functions can receive parameters or values as arguments. It is permitted to nest functions.

IpAdd

`[IpAdd(baseIP, offset1, .., offsetN)]`

Add a series of offsets to a base ip-address.

Offsets may be numeric or an ip-address and will be added to the base to form a new ip-address. Any offset may be a negative number or have negative ip digits. If one offset ip digit is negative, the entire offset is interpreted as negative.

```
[IpAdd(192.168.1.64, 0.0.0.1)] -> 192.168.1.65
[IpAdd(192.168.1.64, -0.0.0.2)] -> 192.168.1.62
[IpAdd(192.168.1.64, 0.0.3.0, 0.0.2.-1)] -> 192.168.2.63
```

Base ip-addresses must be positive ip-addresses, invalid values result in “”.

If base address includes the prefix (eg 192.168.2.1/24) then the resulting address is calculated using the actual start and end addresses of the subnet.

If the sum of the offsets is negative, the result is subtracted from the END of the subnet, and when positive, the result is added to the actual start of the subnet:

```
[IpAdd(192.168.1.64/24, 0.0.0.10)] -> 192.168.1.10
[IpAdd(192.168.1.64/24, -5, 0.0.0.10)] -> 192.168.1.5
[IpAdd(192.168.1.64/24, 0.0.0.-10, -5)] -> 192.168.1.240
```

When the resulting address is not within the subnet, “” is returned.

Invalid offsets and the prefixes /0 and /32 are ignored.

Ipv6Add

`[Ipv6Add(baseIPv6, offset)]`

Calculate an Ipv6 address from a base address and an offset.

The Ipv6 base can include a /prefix which causes the network-address to be normalised to the corresponding size before the offset is added. The offset may have an Ipv6 format (eg ::1 or ::ff) or numeric (0-65536).

If offset have a negative sign included, the offset is subtracted form the net-address, should the net-address include a prefix, the offset is subtracted from the end of the subnet.

Examples:

```
-- addding to a base address:
[Ipv6Add(3001::10, ::1)]     -> 3001::11
[Ipv6Add(3001::20, 10)]      -> 3001::2a
[Ipv6Add(3001::0, ::abba)]   -> 3001::abba
[Ipv6Add(3001::1, 3001::f)]  -> 6002::10

-- adding to a normalised subnet
[Ipv6Add(3001::10/64, ::1)]     -> 3001::1
[Ipv6Add(3001::20/64, 10)]      -> 3001::10
[Ipv6Add(3001::1/64,  3001::f)] -> 6002::f

-- subtracting from a base address
[Ipv6Add(3001::10, -::1)]    -> 3001::f
[Ipv6Add(3001::20, -10)]     -> 3001::16
[Ipv6Add(3001::20, -1000)]   -> 3000:ffff:ffff:ffff:ffff:ffff:ffff:fc38

-- subtracting from a subnet's end address
[Ipv6Add(3001::10/64, -::1)]    -> 3001::ffff:ffff:ffff:fffe
[Ipv6Add(3001::10/64, -0)]      -> 3001::ffff:ffff:ffff:ffff
[Ipv6Add(3001::20/64, -1000)]   -> 3001::ffff:ffff:ffff:fc17
```

Base ip-addresses must be positive ip-addresses, invalid values result in “”.

InvMask

`[InvMask(mask)]`
This inverts an ip-mask in dotted decimal to its complementary value as a cisco wildcard.

Example
The following example:

```
[InvMask(255.255.255.0)]
```

Results in:

```
0.0.0.255
```

NetAddress

`[NetAddress(ip-address, size)]`
This calculates the network address that belongs to the given network size or `prefix`. This is the numeric value that is mainly used instead of the netmask (‘/18’, ‘/23’, etc). The function can be used to calculate the network address of a host ip-address in case the subnet size is known. But the function can also be applied to aggregate subsequent subnets used in an access list.

Example

```
Calculated Supernet = [NetAddress(<Sw_vlan1_addr>,<Ip_supermask>)]
```

NetRange

`[NetRange(ip-address, size)]`
Similar to `NetAddress`, the NetRange function returns the last address of a subnet. Its first argument is an address in a given subnet, the second argument the subnets' size. The size may be specified as the subnet-mask or as its prefix. Using this size, the corresponding subnet-address (the starting address of the subnet) is calculated, and from there its last address.

Example

```
!
Last address = [NetRange(<Net_ip_gateway@vlans>, <net_mask@vlans>)]
!
! or equivalent:
Last address = [NetRange(<Net_ip_gateway@vlans>, <net_size@vlans>)]
!
! The last-but-one address can be obtained using the IpAdd function:
Last -but-one addess = [IpAdd([NetRange(<Net_ip_gateway@vlans>,
<net_size@vlans>)], 0.0.0.-1)]
!
```

Prefix

`[Prefix(mask)]` The Prefix() function returns the prefix from a subnet given its subnet mask.

Mask

`[Mask(prefix)]` The Mask() function returns the subnet mask from a subnet given its prefix

number (0-32)

IpOctet

`[IpOctet(ipaddress, format)]`

Build string from the octets of an ip-address.

The format specifies which octets are concatenated and in what order. Including the '0' in the format indicates each octet is padded with `0` to become 3 digits wide. The default format is '01234'.

examples:

```
ip-based-string: [IpOctet(172.17.0.29, '4')]
results in:
ip-based-string: 29

ip-based-string: [IpOctet(172.17.0.29, '024')]
results in:
ip-based-string: 017029

ip-based-string: [IpOctet(172.17.0.29)]
results in:
ip-based-string: 172017000029

ip-based-string: [IpOctet(172.17.0.29, '04321')]
results in:
ip-based-string: 029000017172
```

Count

`[Count(@context)]`
Returns the number of records in the named context. The argument can be in any of the context forms (`@context`, `param@context`, or `param@context:value`) or even `param@context:column=value`, but the function always returns the number of records in the entire context, not those identified by the parameter or the value.

Example

```
! Number of eVPN Vlans is [Count(@eVPN_vlans)]
|[Count(@eVPN_vlans)]=(1,2)| Yes there are one or two eVPN Vlans
|[Count(@eVPN_vlans)]!=(1,2)| [Error(Expecting one or two eVPN Vlans)]
|[Count(@Ipv6_service)] != 0|{Global_ipv6}   # Load template if not equal to
zero
|[Count(@Ipv4_service:static_route)] != 0|{static_route}   # Load template
if not equal to zero
|[Count(@Ipv4_service:Vlan_id=100)] != 1|[Error("Missing vlan 100")]
```

```
|[count(@ipv4_service:Net_name="loopba??_w*")]|Wildcards are supported. ? = any single character, * = everything.
```

NOTE: commands like '>' or '<' are not possible.

Random

`[Random(min, max , format)]`

Randomly selects a numeric value between the minimum and maximum values provided.

`min` and `max` must be whole positive numbers. Leading zeros in the `min` values indicates zero padding is desired up to the length of `min`. This padding is also controlled using the `format` argument which overrides this behaviour.

The `format` argument can have a numeric value or `Time`. The latter will convert the random value in the `hh:mm:ss` format.

When the `format` is numeric, the value of the format will define the padding width using 0's. A '3' will result in 3 character wide values using zero-padded random values.

Example

```
random number [random(0, 1000)]
random number [random(000, 1000)]
random number [random(0, 1000, 4)]
!
random interval [random(0, 3600, 'time')]
---
Could give the values:
---
random number 31
random number 089
random number 0714
!
random interval 00:29:18
```

List

`[List(separator, parameter@context)]`
Returns a single string consisting of all values of `parameter@context`, each separated with the separator string. This argument can be in any of the context forms (`@context`, `param@context`, or `param@context:'value'`).

Example
The following template:

```
! comma-separated list of all vlan-ids on a port
```

```
switchport trunk allowed vlan [List(‘, ‘, vlan_id@port_subnets)]
! list of vlan-ids on a port
switchport trunk allowed vlan [List( vlan_id@port_subnets)]
```

Will result in this:

```
! comma-separated list of all vlan-ids on a port
switchport trunk allowed vlan 10, 20, 30, 40
! list of vlan-ids on a port
switchport trunk allowed vlan 10 20 30 40
```

Rlist

`[Rlist([separator,[range,]] parameter@context)]`
Returns a single string consisting of all values of `parameter@context`, each separated with the separator string. This argument can be in any of the context forms (`@context`, `param@context`, or `param@context:’value’`).
The separator value is optional and defaults to ' ' (space).
The range value is also optional, but requires the separator value to be presents. The range value defaults to '-'.

The Avaya port name format is also supported. Ports 1/2, 1/3, 1/4 → 1/2-4. A portlist of the form 2, 10, 11, 12, 13, 15 is modified into 2,10-13,15 and 1/2,2/10,2/11,2/12,2/13,3/2,3/3 transforms into 1/2,2/10-12,3/2-3.

Example
The following template:

```
! Ranged list of all the GigabitEthernet interfaces.
interface add [Rlist(Port_name@Interfaces:Gi)]
! Ranged list of vlans on the switch
vlan [Rlist(' and ',' to ', vlan_id@Ipv4_port)]
```

Will result in this:

```
! Ranged list of all the GigabitEthernet interfaces.
interface add Gi00/01-50
! Ranged list of vlans on the switch
vlan 10 to 20 and 30 to 40 and 100
```

Eval

Make sure to test this thoroughly
`[Eval('string')]`
The Eval() function is strictly a perl function, no other code intervenes with the arguments. Eval can be used to calculate with parameters or to execute pattern matching operations.

Numeric compares use ‘==’, ‘>=’ and the like, string compares use ‘eq’, ‘ne’, 'gt' and such. Numerics are never quoted, strings ALWAYS. You can force a number to a string by quoting, but then compare as strings, not numbers.

Note that the use of ‘&&’ is fine, but I would prefer to use the ‘and’ because a ‘||’ will likely cause conflicts with the ‘|…|' of the condition. Using ‘and’ and ‘or’ is easier to read as well.

The string compares ‘eq’, ‘gt’, ‘lt’ and such are tricky because it results in alphanumeric compares where case counts: ‘monkey’ is larger than 'Monkey'.

Example of how to compare with eval

```
# nummeric and string
|[Eval((<Redundant> == 1) && ('<Node_position>' eq 'NA'))]| one goes
#
# force string
|[Eval(('<Redundant>' eq '1') && ('<Node_position>' eq 'NA'))]| two goes
#
# force string >= vs gt (greater then)
|[Eval(<Redundant> >= 0)]| tests as greater
|[Eval('<Redundant>' gt '0')]| tricky
#
# string gt (greater then)
|[Eval('monkey' eq 'Monkey')]| not the same
|[Eval(lc 'monkey' eq lc 'Monkey')]| the same after lowercase compare
#
|[Eval('monkey' gt 'Monkey')]| monkey wins over Monkey
#
```

Example of the Eval flexibility

```
[Eval((<param>+100)*2)]
[Eval('<param>' =~ /^zt/i)]
[Eval(<param>>= 1024)]
[Eval(substr(<param>,0,2))]
|[Eval((<Redundant> == 1) or (<MigrationPhase> == 28))]|access-list 60
```

Example of how NOT to use Eval

```
# mismatch nummeric
|[Eval('<Redundant>' == 1)]| num mismatch one
|[Eval(<Redundant> == '1')]| num mismatch two
#
# mismatch string
|[Eval('<Node_position>' eq NA)]| str mismatch one
|[Eval(<Node_position> eq 'NA')]| str mismatch two
#
|[Eval('1' ne '01')]| oops
|[Eval('1' gt '01')]| oops again
#
```

The `Eval()` function can be used in conditions. In that case the true or not true results are evaluated as 1 and 0. In these cases it's useful to explicitly validate these values:

```
|[Eval('<param>' =~ /^zt/i)]='1'| template line
```

```
!
My Site type is: <Site_Type>
!
-- when string must match a pattern use:
| [Eval('<Site_type>' =~ /cpe/i)] =1 | <Site_type> like CPE
-- note the quotes around the *string* <Site_type>, the substitution takes
place BEFORE the functions like Eval()
!
-- This works identical to:
| [Eval('<Site_type>' =~ /cpe/i)] | <Site_type> like CPE
!
-- when it must NOT match the pattern use
| [Eval('<Site_type>' =~ /cp/i)] !=1 | <Site_type> not like CPE
| [Eval('<Site_type>' =~ /core/i)] !=1 | <Site_type> not like CORE
!
-- the !~ will (currently) not work, it produces a syntax error:
| [Eval('<Site_type>' !~ /cp/i)] | <Site_type> not like CPE
!
```

RESULT:

```
Configuration encountered errors
!
My Site type is: CPE
!
    CPE like CPE
!
    CPE like CPE
!
    CPE not like CORE
!
|!(XXXXX Cannot evaluate: ''CPE' ~ /cp/i': syntax error at (eval 14) line 1,
near "'CPE' ~"
)| CPE not like CPE
!
```

Error

`[Error(message)]`
This generates an error conditions at the moment of template generation. This function is meant to be able to test conditions that should limit the roll-out of the template. As an example, one can count with the `Count()` function if the number of vlans are correct. If not, that the `Error()` will generate a descriptive error message.

Example

```
|[Count(@Port_subnets)] = '0'|[Error(‘No subnets assigned to this port’)]
```

MD5

`[MD5(string)]`
This calculates a MD5_hex hash over the argument string. This function can be used to encrypt information like a password that should be exchanged between two nodes.

Example

```
exchange password [MD5(‘<hostname>:mypassword’)]
```

Dec_hex

`[Dec_hex(string)]`
`[Dec_hex(string, padding)]`
Converts a Decimal into its Hexadecimal value.

The optional padding value specifies the length of the value returned by adding leading '0' when needed. If the padding value is negative, the padding creates trailing spaces.

Hex_dec

`[Hex_dec(string)]`
`[Hex_dec(string, padding)]`
Converts a Hexadecimal into its (unsigned) Decimal value.

The optional padding value specifies the length of the value returned by adding leading '0' when needed. If the padding value is negative, the padding creates trailing spaces.

Str_hex

`[Str_hex(string)]`
Creates a hexadecimal code from a string

Hex_str

`[Hex_str(hex-string)]`
Converts a hexidecimal code back to its corresponding string

Ip_hex

```
[Ip_hex(<ip_address>)]
[Ip_hex(<ip_address>, padding)]
```

Convert a dotted-decimal IPv4 address in to a Hexadecimal value. It converts each decimal value into a hex value and concatenates those in one string. Bij default each converted decimal is padded to 2 characters, adding a leading '0' where needed.

The optional padding variable can be increased to create more leading '0's.

```
[Ip_hex('10.141.61.171')]
results in:
  0A8D3DAB

[Ip_hex('10.141.61.171', 4)]
results in:
  000A008D003D00AB
```

Hex_ip

```
[Hex_ip(<ip_in_hex>)]

[Hex_ip(<ip_in_hex>, padding)]
```

Convert a hexadecimal formatted IPv4 address back into an IP value. It converts each hexadecimal value into a decimal value and concatenates those in a dotted decimal string.

Bij default the hexadecimal string is chopped into sets of 2 characters before converting those to decimal values. By increasing the padding value, the number of characters grouped before conversion can be increased.

The function returns a string of dotted decimals, but does not check if it forms a valid ip-address!

```
[Hex_ip('0A8D3DAB')]
results in:
  10.141.61.171
[Hex_ip('000A008D003D00AB', 4)]
results in:
  10.141.61.171
```

Null

The Null function will not be used a lot in templates. It serves to be able to cancel the line with a specific function from the configuration. It’s therefor mainly used when the value `[Null]` is generated indirectly by parameter substitution in order to suppress that line. Internally, the Null-

function is used when filling the extra fields (parameters) that result during a 'Transform' of a Relation-query. Regarding transformed parameters for which no value exists the `[null]` is being substituted so only existing values can be used.

Replace

`Replace(string,match,[replace],[all])`

Replace searches for a specific string-value and replaces it with another string-value. The replace value can be empty. The match string is non case-sensitive.

The optional `all` argument, when non-zero, will replace the string not just for the first occurrence, but for all occurrences

Example
The following

```
<hostname> - [Replace(<hostname>,'test', 'TEMP')]
<hostname> - [Replace(<hostname>,'test')]

<hostname> - [Replace(<hostname>,'t','x')]
<hostname> - [Replace(<hostname>,'t','x',1)]
```

Will result in:

```
TEST_ROUTER001 - TEMP_ROUTER001
TEST_ROUTER001 - ROUTER001

TEST_ROUTER001 - xEST_ROUTER001
TEST_ROUTER001 - xESx_ROUxER001
```

Ucase / Lcase / FirstCap

`Ucase(string)`
`Lcase(string)`
`FirstCap(string)`

These functions convert the case of the string, where Ucase stands for uppercase, Lcase for lowercase and FirstCap stands for First character Capital.

Example
The following

```
<hostname> - [Ucase(<hostname>)]
<hostname> - [Lcase(<hostname>)]
<hostname> - [FirstCap(<hostname>)]
```

Will result in:

```
test_router001 - TEMP_ROUTER001
TEST_ROUTER001 - test_router001
test_router001 - Test_router001
```

Substring

`Substring(string, offset, length)`

Substring extracts a portion of a string based on startpoint and length.
First character is at offset zero.

If OFFSET is negative, starts that far back from the end of the string.
If LENGTH is omitted, returns everything through the end of the string.
If LENGTH is negative, leaves that many characters off the end of the string.

```
my $s = "The black cat climbed the green tree";
[Substring("The black cat climbed the green tree", 4, 5)]      # black
[Substring("The black cat climbed the green tree", 4, -11)]    # black cat
climbed the
[Substring("The black cat climbed the green tree", 14)]        # climbed the
green tree
[Substring("The black cat climbed the green tree", -4)]        # tree
[Substring("The black cat climbed the green tree", -4, -2)]    # tr
```

WordIdx

`WordIdx(string, separator, index, index, …)`

WordIdx splits a string using the 'separator' and returns the Word indicated by the index number. The default separator is a space but the separator can be a regular expression like

```
# split on comma with or without whitespace
'\s*,\s*'
# split on the dot character (of an ip-address)
'\.'
```

Use quotes around the separator when a comma is used, and use a backslash \ to protect special characters used in regular expressions (`'.', '[', ']', '^', '$', '-', '+', '(', ')'`)

When the index is omitted, the first word is returned by default. Use 1 for the first word, 2 for the second and so on. An index of -1 returns the last, -2 the fore last and so on.

If multiple indices are used, the corresponding words are returned in the order requested. Multiple indexed words are always separated by a space.

The index `0` has special meaning, it returns the number of elements that the split produced.

Some examples and their results:

```
#
[WordIdx('one two three')]
=> one
#
[WordIdx('one two three',,2)]
=> two
#
[WordIdx('one two three',,-1,1)]
=> three one
#
[WordIdx('one, two, three', '\s*,\s*', 3)]
=> three
#
[WordIdx('192.168.17.1', '\.', 4,3)]
=> 1 17
#
[WordIdx('192.168.17.1', '\.', 4)].[WordIdx('192.168.17.1', '\.', 3)]
=> 1.17
#
[WordIdx('192.168.17.1', '\.', 0)]
=> 4
#
```

RowIdx

```
RowIdx(parameter@relation, row)
RowIdx(parameter@relation:value, row)
```

RowIdx returns a single parameter value from a specific row of a relation instead of all the parameter values of each row.

```
Port-name: <port_name@interfaces>
-> this will insert all interface names
#
RowIdx: [RowIdx(port_name@interfaces, 0)]
-> this will insert the first interface name in the relation
#
RowIdx: [RowIdx(port_name@interfaces:giga*, -2)]
-> this will insert the last but one gigabit interface name
```

When accessing parameters through relations, all (matching) parameter values of the relation are inserted. But occasionally, only the first parameter value of the relation is desired. Or only the last.

For these situations the function **RowIdx()** is created. It takes a parameter reference from a relation and the row number (index) as arguments. Only one value is substituted.

The relation reference may have a value filter and supports wildcards. By default row '0' is used.

Row index numbers can be positive (0 at top), or negative (-1 at bottom). Non existent rows will result in an empty string.

Coalesce

`Coalesce(<parameter>, <parameter@relation>, …)`

Coalesce() returns from a list of parameters the first non-empty value. There is no practical limit on the number of parameters in the list. Undefined parameters and empty strings are both considered an empty value, but the number zero ('0') is considered a non-empty value. If none of the parameters contains a value. the function returns a blank (undefined).

The Coalesce function is intended to insert a parameter value that can be defined in different places (objects) but have a hierarchical dependency. By listing each in the hierarchical order in the coalesce function, the one considered overriding the remaining is returned - if available. Otherwise the next one will be considered.

```
Enable-secret: [coalesce(<Enable_secret@SiteRouter>,
<Default_enable_secret@Domain>, <Enable_secret@Lookup)]

-> Enable-secret: scramble-03427
```

All parameters in the Coalesce list must be enclosed in edged brackets ('<..>') or the parameter name will be returned as entered.

Note that when using relations, the line including the Coalesce() will be executed on all rows the relation returned.

Node_mgmt_ipv4

`Node_mgmt_ipv4(<hostname>)`

Returns the ipv4 management address of the node indicated. The node must be known to the YCE database

Node_mgmt_ipv6

`Node_mgmt_ipv6(<hostname>)`

Returns the ipv6 management address of the node indicated. The node must be known to the YCE database

Node_port_ipv4

`Node_port_ipv4(<hostname>, <portname>, <columnname>, [<filter>])`

The functions **Node_port_ipv4** and **Node_port_ipv6** allow you to retrieve ip-subnet/ipv6-subnet attributes from subnets assigned to a specific port from a node. Three mandatory arguments are required: the 'nodename', the 'portname' and the ip-subnet 'columnname'. The portname can either be the internal Port_name or the vendor dependent full interface name.

The optional fourth argument 'filter' can be used to select the desired subnet if more than one subnet is assigned to the port. This 'filter' option behaves similar to the relation-syntax filtering where a 'value' by itself will be compared against all columns (like “column@Ip_subnet:filter”) or a 'column=value' filter will compare only the indicated column (like “column@Ip_subnet:key=value”).

Examples:

```
[Node_port_ipv4(<hostname>,Gi01/00/03,Ip_parameter)]
[Node_port_ipv4(<hostname>,GigabitEthernet1/0/3,Ip_parameter)]

[Node_port_ipv4(<hostname>, 'Gi01/00/03', 'Net_address', 'Vrf_id=3')]
[Node_port_ipv4(<hostname>, 'GigabitEthernet1/0/3', 'Net_address', Vrf_id =
'3')]
```

Node_port_ipv6

`Node_port_ipv6(<hostname>, <portname>, <columnname>, [<filter>])`

The functions **Node_port_ipv4** and **Node_port_ipv6** allow you to retrieve ip-subnet/ipv6-subnet attributes from subnets assigned to a specific port from a node. Three mandatory arguments are required: the 'nodename', the 'portname' and the ip-subnet 'columnname'. The portname can either be the internal Port_name or the vendor dependent full interface name.

The optional fourth argument 'filter' can be used to select the desired subnet if more than one subnet is assigned to the port. This 'filter' option behaves similar to the relation-syntax filtering where a 'value' by itself will be compared against all columns (like “column@Ip_subnet:filter”) or a 'column=value' filter will compare only the indicated column (like “column@Ip_subnet:key=value”).

The ipv6 address will be returned in the 'short' format.

Examples:

```
[Node_port_ipv6(<hostname>,Gi01/00/03,Ip_parameter)]
[Node_port_ipv6(<hostname>,GigabitEthernet1/0/3,Ip_parameter)]

[Node_port_ipv6(<hostname>, 'Gi01/00/03', 'Ipv6_subnet_prefix', 'Vrf_id=3')]
[Node_port_ipv6(<hostname>, 'GigabitEthernet1/0/3', 'Ipv6_subnet_prefix',
Vrf_id = '3')]
```