Conditionals

Frequently parameter substitution in the templates require the conditional inclusion of the command line. Various forms of conditionals are supported by the YCE syntax. In its basic form the condition is stated between two vertical bars `|<condition>|` preceding the command line.

Please note that in case of setting a login banner containing the vertical bar using a template, the following will be parsed as a condition and therefore not result in the envisioned banner.

This template line

```
banner motd d |------- Warning: THIS IS A PRIVATE COMPUTER SYSTEM -------| d
```

Will result into this

```
banner motd d  d
```

However, by protecting the bars with a backslash will result in the desired command:

```
banner motd d \|------- Warning: THIS IS A PRIVATE COMPUTER SYSTEM -------\|
d
```

Substitution condition

```
|<parameter>| template line
|’value’| template line
|’’| template line
|[Function()]| template line
```

This tests the for a value of `<parameter>`. In case the parameter is not blank (empty string), then the configuration line is parsed and inserted. If the parameter is empty (’’) then the template line is skipped. In case the parameter does not exist in the current context, then an error message is generated.

```
|!<parameter>| template line
|!’value’| template line
```

This form reverses the behavior: if the parameter is empty, the line is included. When it turns out it doesn’t exist, an error message will appear.

In all cases the `template line` can include parameters from default or named contexts. It is quite common to test for a value before including the line based on that value:

```
|<local_user>| username <local_user> secret 0 <local_passwd>
```

Compare condition

```
|<parameter> = ‘value’| template line
|<parameter> = <parameter2>| template line
|<parameter> = ’’| template line
|[Function()] = ...| template line
```

These compare the value of <parameter> with the mentioned values and then add the template line conditionally. The comparison is always made between the strings, even in case of numeric values. The results are analog to the conditional substitution.

Also here it is possible to have ‘not equal to’. The negation ! can can be placed at the start of the condition or being contracted with the =. Both forms behave equally.

```
|<parameter> != ‘value’| template line
| ! <parameter> = value| template line
```

Quotes are optional for fixed values, but is recommended to use them. Actually the < > around parameters are also optional, although this form is strongly recommended. In many cases, the correct interpretation is given to the forms with or without quotes and/or brackets. See the examples:

```
|<hostname>| results in this line
|'hostname'| results in this line
|<hostname> = <hostname>| results in this line, interpretation is
unambiguous
|<hostname> = hostname| no results
|<hostname> = 'hostname'| no results, interpretation is unambiguous
```

```
|<hostname> = 'hvs-rn06001'| results in this line, it’s the real hostname
|<hostname> = hvs-rn06001| results in this line, interpretation is correct,
but ambiguous
|hvs-rn06001 = hostname| no results, actually not correct
|hvs-rn06001 = <hostname>| also no results, incorrect
|'hvs-rn06001' = <hostname>| results in this line interpretatie is
unambiguous
```

Multiple comparison

```
|<parameter> = (‘value1’,’value2’,...,'valueN')| template line
|<parameter> = (<parameter2>,<parameter3>,...,'valueN')| template line
```

To prevent the same conditional template line to be used for a multiple number of values of a parameter, the values can be specified as a list. The brackets are optional. Also negations (! and !=) are possible.

Evaluation

The all round `[Eval()]` function is very much suited for numeric comparisons and substring conditions. See Eval function for more details.

Multiple comparison with context

```
|<parameter@context>| template line
|<parameter@context> = <parameter2>| template line
|<parameter@context> = ‘value’| template line
```

This generates a conditional line for all values of the context. This is mainly useful in the = equation to use the corresponding index(es) of the context for the template line.

```
|<parameter@context> = (‘value1’,’value2’,...)| template line
|<parameter1@context> = <parameter2@context>| template line
```

This equation has on both sides of the = sign multiple values that are compared with each other. For every match (non case sensitive string comparisons) a template line is generated. The line shall use the same indexes in the context while substituting parameters as used for the conditions.

```
|<parameter> = (<parameter2@context>)| template line
```

AND conditionals

It is possible to create an **AND** construction with the conditionals.

By using:

```
|<valueA> = 'test'||<valueB> = 'address'|cli command
```

The cli command will only be executed when *Conditional A* **AND** *Conditional B* are true.
Likewise:

```
|<valueA> = 'test'| command one |<valueB> = 'address'| command two
```

will only result in any output when both conditionals are true. When one of the conditionals fails, none of commands will be executed.

Repeat and Else condition

The result of the last conditional can be reused by testing on an empty condition (`||`). When combined with the negation (`!`) the last conditional result can be used to create an “ELSE” conditional: `|!|`:

```
|<site_type> = 'retail' | {retail_stuff}

add some generic config lines

|!| {non_retail_stuff}
```

Be aware that the conditionals are evaluated at run-time. The LAST conditional result is really the last one executed and is not aware of its context. If, like in the example above, a sub-template is included that itself uses conditionals, then these will likely alter the ||.

The “else” (|!|) will NOT be affected by possible conditionals in a sub template, it is not executed. The above example is quite safe.

The repeat conditional is therefore best used for simple in-line commands or blocks.

```
# UNSAFE:
# if {retail_stuff} uses conditionals
|<site_type> = 'retail' | {retail_stuff}

# because || now represents the result of the last conditional in
{retail_stuff}
|| {more_retail_stuff}

# SAFE:
|<site_type> = 'retail' | retail command line one \
retail command line two \

do generic commands

# now reuse the last conditional safely:
|| {more_retail_stuff}

# but at this position, the ''||'' or ''|!|'' result might have changed
```