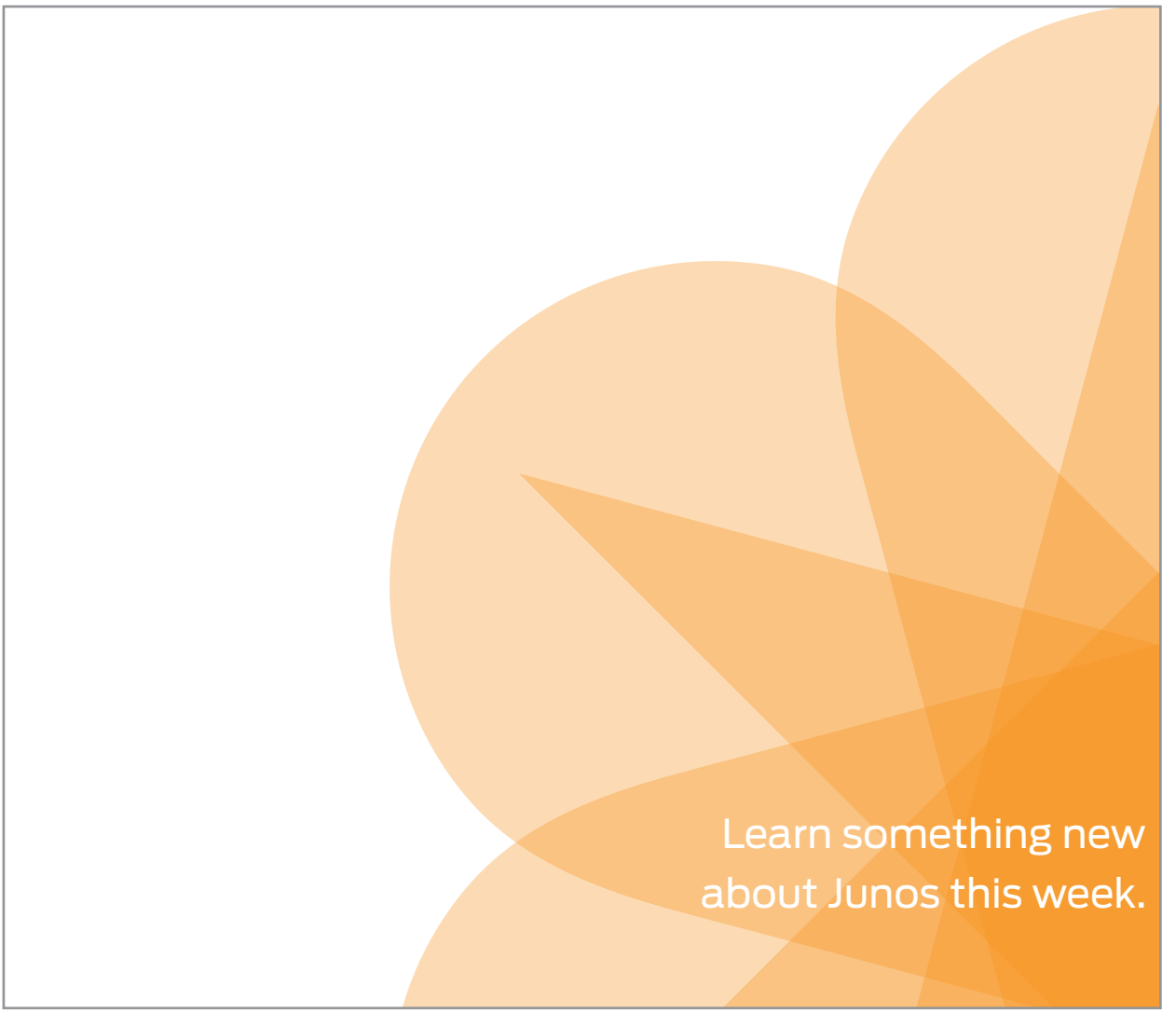
Junos® Automation Series

THIS WEEK:
JUNOS AUTOMATION REFERENCE FOR SLAX 1.0



By Curtis Call

THIS WEEK: JUNOS AUTOMATION REFERENCE FOR SLAX 1.0

SLAX is a syntax overlay of the XSLT programming language. While XSLT is used internally by Junos to power its on-box scripting capabilities, it is not the most intuitive or efficient of languages, so SLAX was created to simplify on-box script programming and make it more comfortable to write. This reference guide provides descriptions and examples of all of the basic components of the SLAX language including its statements, operators, functions, elements, templates, and default parameters.

Only SLAX 1.0 is covered in this volume. Even though the latest version of SLAX is now 1.1, and the newer version provides a number of enhancements over version 1.0, version 1.1 is not available in Junos yet. Instead, this guide focuses exclusively on the components available in the SLAX 1.0 language.

This Week: Junos Automation Reference with SLAX 1.0 specifically covers writing SLAX scripts for the Junos operating system. The SLAX language can be used to perform generic XML transformations in other operation systems, but this guide assumes that the SLAX script is being processed as a Junos on-box script.

"This book is simply awesome. I used it twice the very first day I had it! It's a Swiss Army knife of a reference, all the Junos automation a user might need in one monstrously-detailed package."

Phil Shafer, Junos UI Architect, Juniper Networks

THE DEFINITIVE REFERENCE FOR JUNOS AUTOMATION WITH SLAX 1.0:

- Includes all the statements that are available in SLAX 1.0, a demonstration of their syntax, and examples of each.
- Contains all the boolean, comparison, mathematic, and other operators in the SLAX language, including those derived from XPath, those added to improve efficiency, and certain scenarios where the SLAX-specific operators are not supported.
- Documents all of the functions that are available natively in SLAX 1.0, including those functions that come from XPath, XSLT, EXSLT, and Junos itself.
- Details the XSLT elements and additional extension elements that might be needed within a Junos SLAX script.
- Documents the templates that are available in the junos.xsl import file as well as the default global parameters.

Published by Juniper Networks Books
www.juniper.net/books

ISBN 978-1936779345
52800
9 781936 779345
07500215

JUNIPER NETWORKS

Junos® Automation Series

This Week:
Junos Automation Reference for SLAX 1.0

by Curtis Call

JUNIPER NETWORKS

Published by Juniper Networks Books
Writer: Curtis Call
Editor in Chief: Patrick Ames
Proofreader: Nancy Koerbel
J-Net Community Management: Julie Wider

About the Author
Curtis Call is a Senior Systems Engineer at Juniper Networks, has over a decade of experience working with Junos, and has authored multiple books on Junos on-box automation. He is a Juniper Networks Certified Internet Expert (JNCIE-M #43).

Author's Acknowledgment
The author would like to thank all those who helped in the creation of this reference book.

This book is available in a variety of formats. For more information see www.juniper.net/dayone.

Send your suggestions, comments, and critiques by email to: dayone@juniper.net.

Version History: v1 (This Week) July 2011

ISBN: 978-1-936779-34-5 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-936779-35-2 (ebook)

2 3 4 5 6 7 8 9 10 #7500215-en

Juniper Networks Books are singularly focused on network productivity and efficiency. See the complete library at www.juniper.net/books.

Welcome to *This Week*

This Week books are an outgrowth of the extremely popular *Day One* book series published by Juniper Networks Books. *Day One* books focus on providing just the right amount of information that you can do, or absorb, in a day. On the other hand, *This Week* books explore networking technologies and practices that in a classroom setting might take several days to absorb. Both book series are available from Juniper Networks at: www.juniper.net/dayone.

This Week is a simple premise – you want to make the most of your Juniper equipment, utilizing its features and connectivity – but you don't have time to search and collate all the expert-level documents on a specific topic. *This Week* books collate that information for you, and in about a week's time, you'll learn something significantly new about Junos that you can put to immediate use.

This Week books are written by Juniper Networks subject matter experts and are professionally edited and published by Juniper Networks Books. They are available in multiple formats, from eBooks to bound paper copies, so you can choose how you want to read and explore Junos, be it on the train or in front of terminal access to your networking devices.

About this Reference Book

SLAX is a syntax overlay of the XSLT programming language. XSLT, while it is used internally by Junos to power its on-box scripting capabilities, is not the most intuitive or efficient of languages, so SLAX was created to simplify on-box script programming and make it more comfortable to write. This reference guide provides descriptions and examples of all of the basic components of the SLAX language: including its statements, operators, functions, elements, templates, and default parameters.

Only SLAX 1.0 is covered in this volume. The latest version of SLAX is now 1.1, and the newer version provides a number of enhancements over version 1.0; however, as of the time of this writing, version 1.1 is not available in Junos, and it will not be discussed within this reference guide; instead, this guide focuses exclusively on the components available to the SLAX 1.0 language.

This guide covers writing SLAX scripts specifically for the Junos operating system. The SLAX language can be used to perform generic XML transformations in other operation systems as well, but the capabilities and caveats discussed in this guide are based on the assumption that the SLAX script is being processed as a Junos on-box script and may not apply in those other situations.

The material enclosed is current as of Junos 11.1, released in the first half of 2011. Each topic description includes the minimum Junos version required to use the SLAX component, which refers to the first Junos release that contained support for the functionality in its R1 revision. At times, support was actually initially introduced for SLAX features in a post-R1 revision (R2, R3, etc.) of an earlier Junos release than indicated, but those exceptions are not documented in this guide. Also, because SLAX was officially released in Junos 8.2, that is the absolute minimum version reported in this guide; however, many of the non-SLAX-specific components were present in Junos prior to that release for use with XSLT scripts.

Professional Acknowledgements

Daniel Veillard and the other programmers that created the libxslt library deserve recognition and commendation for the superb open source XSLT processor they developed and continue to maintain. This library is used as the underlying script engine for Junos SLAX scripts, and without it Junos might not have any scripting capabilities to write a book about.

I'd also like to acknowledge Phil Shafer and his fellow Juniper UI developers who gave us the SLAX scripting language, making it more efficient and enjoyable to write on-box Junos scripts. If not for this innovation it is doubtful that Junos scripts would have achieved the level of acceptance that they have.

Curtis Call
July 2011

Table of Contents

Part 1: Statements

Part 2: Operators

Part 3: Functions

Part 4: Elements

Part 5: Templates

Part 6: Default Parameters

Appendices

Part 1: Statements

The SLAX scripting language was created to provide an efficient and comfortable alternative to XSLT, which is the language used by the Junos on-box script processor. Much of this improvement stems from the replacement of many XSLT elements with more programmer-friendly keyword statements. These look and are structured in similar ways to common programming languages, which allows programmers to better leverage their programming knowledge and experience as they learn how to create Junos on-box scripts.

This section documents all of the statements that are available in SLAX 1.0. A demonstration of their syntax is shown along with an in depth description of their uses and caveats. In addition, an example is provided with each statement, showing how it can be used in an actual SLAX script.

apply-templates

Minimum Version: Junos 8.2

Syntax

apply-templates*;*
apply-templates *node-set-expression;*
apply-templates *node-set-expression {*
...optional mode...
...optional sort elements...
...optional parameters...
}

Description

The **apply-templates** statement is used to invoke match templates for its selected node list, which is derived from its node-set expression, or if no expression is included then the node list consists of all child nodes of the current node.

Code example:

```
apply-templates;
```

Selected node list:

Child nodes of the current node

Code example:

```
apply-templates interface;
```

Selected node list:

<interface> child nodes of the current node

The **apply-templates** statement iterates over each node in its node list, selecting the preferred match template that has a matching pattern for the node and invoking that template with the node as the current node and the selected node list as the current node list. A code block can be included, enclosed in curly brackets, if **<xsl:sort>** instruction elements, or **mode** or **with** statements should be included.

By default, the current node list is arranged with the nodes in document order; however, **<xsl:sort>** instruction elements can be added within the **apply-templates** code block to instruct the script processor to sort the node list in a different manner:

Code example:

```
var $configuration = jcs:invoke( "get-configuration" );
apply-templates $configuration/interfaces/interface {
    <xsl:sort select="name">;
}
```

In the above example, <interface> nodes will be processed by the script processor based on the comparison of their <name> child elements rather than on their document order.

If parameter values should be specified for the invoked match template, then these values must be assigned within the **apply-templates** statement's code block, using the **with** statement:

```
apply-templates route {
    with $next-hop = "10.0.0.1";
}
```

When multiple parameters are assigned, their position relative to each other is insignificant because the template parameters are assigned based on their name rather than their position:

```
apply-templates route {
    with $next-hop = "10.0.0.1";
    with $metric = "10";
}
apply-templates route {
    with $metric = "10";
    with $next-hop = "10.0.0.1";
}
```

There is no need to specify a value for a parameter if the parameter is being assigned the value of a parameter or variable that shares the same name within the calling template; instead, when the parameter name is present in the **apply-templates** statement, but lacks an assignment, SLAX automatically assumes that the parameter's value should be set to the value of the variable or parameter of the same name within the calling template. In other words, the following two match template calls are identical:

```
var $string = "Junos";
apply-templates message {
    with $string = $string;
}
apply-templates message {
    with $string;
}
```

However, when assigning parameter values in this manner, it is still necessary to include the parameter name, without an assigned value, in the **apply-templates** statement; otherwise, if the parameter is not specified as part of the **apply-templates** statement then it is assigned its default value.

A mode can be indicated for the invoked templates by including the **mode** statement within the **apply-templates** code block. See the description of the **mode** statement for details on how this impacts the match template selection.

By default, the results of the template call are written to the result tree, but they can be redirected into a variable instead if the **apply-templates** statement is enclosed within curly brackets and the entire code block is assigned to a variable:

```
var $configuration-changes = { apply-templates route; }
```

This results in a result tree fragment variable. If a node-set variable is desired instead, so that location paths can be used to extract its contents, then the node-set conversion operator should be used instead of the assignment operator:

```
var $configuration-changes := { apply-templates route; }
```

In the above code example, $configuration-changes is a node-set because it was assigned through the node-set conversion operator ":=" rather than the assignment operator "=".

The **apply-templates** statement can only be used within a code block. It cannot be used at the top-level of the script.

Other than the starting boilerplate template, match templates are not commonly used within Junos SLAX scripts; instead, most programmers modularize their code with named templates or custom functions, so the **apply-templates** statement is rarely used.

Example

This op script demonstrates how the **apply-templates** statement can call a match template to process each node in the selected node list.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve storage information */
        var $storage = jcs:invoke( "get-system-storage" );
        var $pattern = "%-15s %10s %10s";
        /* Display the filesystems, sorted based on their capacity & size */
        <output> "Filesystems, based on capacity:";
        <output> jcs:printf( $pattern, "Name", "Capacity", "Size" );
        apply-templates $storage/filesystem {
            /* Provide the pattern */
            with $pattern;
            /* Primary sort is the capacity */
            <xsl:sort select="used-percent" data-type="number" order="descending">;
            /* Secondary sort is the total size */
            <xsl:sort select="total-blocks" data-type="number" order="descending">;
            /* Third sort is the name */
            <xsl:sort select="filesystem-name">;
        }
    }
}

match filesystem {
    param $pattern;
    <output> jcs:printf( $pattern, filesystem-name, concat( used-percent, "%"),
        total-blocks/@junos:format );
}
```

Output

```
Filesystems, based on capacity:
Name              Capacity       Size
/dev/md0              100%       360M
procfs                100%       4.0K
devfs                 100%       1.0K
devfs                 100%       1.0K
devfs                 100%       1.0K
/cf                    52%       293M
/dev/da0s2a            52%       293M
/cf/var/jail           11%       342M
/cf/var/log            11%       342M
/dev/bo0s3f            11%       342M
/dev/md1               10%       168M
/dev/md2                0%        39M
/dev/bo0s3e             0%        24M
/dev/md3                0%       1.8M
```

call

Minimum Version: Junos 8.2

Syntax

call *template-name;*
call *template-name(...optional parameters...);*
call *template-name {*
...optional parameters...
}

Description

The **call** statement is used to invoke a named template. Values for the template's parameters can be provided as the template is called; otherwise, they will be assigned their default value. The **call** statement supports multiple formats, but in all cases it includes the template-name, which must be a valid qualified name (see Appendix B), and must be hardcoded into the script because it is not possible to specify the template's name through a variable.

If a template named display-user-name required no parameters, then it could be called in this manner:

```
call display-user-name;
```

However, if parameter values should be specified, then these values must be assigned either within parentheses, or within the **call** statement's code block, using the **with** statement:

```
call to-upper( $string = "Junos" );

call to-lower {
    with $string = "Junos";
}
```

When multiple parameters are assigned, their position relative to each other is insignificant because the template parameters are assigned based on their name rather than their position:

```
call build-string( $length = 10, $character = "!" );
call build-string( $character = "!", $length = 10 );
```

There is no need to specify a value for a parameter if the parameter is being assigned the value of a parameter or variable that shares the same name within the calling template; instead, when the parameter name is present in the **call** statement but lacks an assignment, SLAX automatically assumes that the parameter's value should be set to the value of the variable or parameter of the same name within the calling template. In other words, the following two template calls are identical:

```
var $string = "Junos";
call to-upper( $string = $string );
call to-upper( $string );
```

However, when assigning parameter values in this manner, it is still necessary to include the parameter name, without an assigned value, in the **call** statement; otherwise, if the parameter is not specified as part of the **call** statement then it is assigned its default value.

By default, the results of the template call are written to the result tree, but they can be redirected into a variable instead if the **call** statement is enclosed within curly brackets and the entire code block is assigned to a variable:

```
var $capitalized-string = { call to-upper( $string ); }
```

This results in a result tree fragment variable. If a node-set variable is desired instead, so that location paths can be used to extract its contents, then the node-set conversion operator should be used instead of the assignment operator:

```
var $results := {call jcs:load-configuration( $connection, $configuration);}
```

In the above code example, $results is a node-set because it was assigned through the node-set conversion operator ":=" rather than the assignment operator "=".

The **call** statement can only be used within a code block. It cannot be used at the top-level of the script.

Example

This op script demonstrates different ways to use the **call** statement to invoke a named template. It also shows how default parameter values can be applied if values are not provided for all of the template's parameters.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> {
            /* Rely on default values for $length and $index */
            call insert-filler( $string = "SLAX", $character = " " );
        }
        <output> {
            var $string = "Hello World";
            call insert-filler {
                with $string;
                with $index = 2;
                with $length = 10;
                with $character = "_";
            }
        }
    }
}

/* Inserts variable number of filler characters at specified position within string */
template insert-filler() {
    param $string;
    param $index = string-length( $string ) div 2;
    param $length = 1;
    param $character = "-";

    var $before = substring( $string, 1, $index );
    var $after = substring( $string, $index + 1 );

    expr $before _ str:padding( $length, $character ) _ $after;
}
```

Output

```
SL AX
He__________llo World
```

comment

Minimum Version: Junos 8.2

Syntax

comment *"comment string";*

Description

The **comment** statement generates a XML comment node, assigning the statement's string value to the comment, and inserts the node into the result tree. This statement cannot be used at the top-level of a script. It can only appear within a code block.

The comment string cannot contain the character combination "--" anywhere within the string, and it also cannot end with the character "-".

This statement is rarely used because Junos processes the result tree directly, so any added XML comments usually serve no purpose; however, adding comment nodes to XML documents that are written to the disk through the **<xsl:document>** instruction element might be beneficial. This is demonstrated in the example script for this statement.

The **comment** statement cannot be used to add a comment to the Junos configuration through a configuration change; instead, use the <junos:comment> element for that purpose.

Code example:

```
var $configuration = {
    <configuration> {
        <system> {
            <junos:comment> "This creates a configuration comment";
            <host-name> "srx210h";
        }
    }
}
var $connection = jcs:open();
var $results := {call jcs:load-configuration( $connection, $configuration );}
```

Output:

```
jnpr@srx210> show configuration system host-name
/* This creates a configuration comment */
host-name srx210h;
```

Example

This op script demonstrates how the **comment** statement can be used to add a comment to a XML document that is written to the disk by the **<xsl:document>** instruction element.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Strip whitespace so it indents well when written to disk */
strip-space user-context;

match / {
    <op-script-results> {

        /* Write <user-context> node to disk */
        <xsl:document href="/var/tmp/user-info" indent="yes"> {

            var $users = jcs:invoke( "get-system-users-information" );
            var $entry = $users/uptime-information/user-table/
                user-entry[ tty == substring-after( $junos-context/tty, "/dev/tty" ) ];

            /* Indicate the login time via a comment */
            comment "Login time: " _ $entry/login-time;
            /* Copy user-context into the document */
            copy-of $junos-context/user-context;
        }
    }
}
```

Output

```
jnpr@srx210h> file show /var/tmp/user-info
<?xml version="1.0"?>
<!--Login time: 12:03PM-->
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user-local</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

copy-of

Minimum Version: Junos 8.2

Syntax

copy-of *expression;*

Description

The **copy-of** statement copies the result of its expression into the result tree (which could be redirected into a result tree fragment variable). The expression is a XPath expression, including full support for most SLAX-

specific operators, but the SLAX string concatenation operator cannot be used at the time of this writing. (The **concat**() function could be used instead). This statement cannot be used at the top-level of a script. It can only appear within a code block.

The behavior of the **copy-of** statement depends on the data type of the expression result:

- result tree fragment – The complete XML contents of the fragment are copied into the result tree.
- node-set – Each node from the node-set is copied into the result tree.
  - Element nodes are copied along with their attributes, namespaces, and child nodes, including text nodes.
  - Root nodes are copied by copying their child nodes. The root node itself is not copied.
  - A deep copy of element nodes is performed, so the copying process will repeat for each child element node, and each child element node of those child element nodes, etc. until the complete hierarchy has been copied.
- All other data types – The result is converted into a string, which is inserted into the result tree
  - At the time of this writing, the external data type (stored expressions) are not supported with the **copy-of** statement.

One common use of the **copy-of** statement is to copy the results of a Junos CLI command into the result tree so that Junos will parse the XML data and display the command output in the same way as it appears in the CLI. This makes it possible to show the results of multiple CLI commands through a single script:

Example code:

```
<op-script-results> {
    <output> "show version:";
    var $result1 = jcs:invoke( "get-software-information" );
    copy-of $result1;
    <output> "show system users:";
    var $result2 = jcs:invoke( "get-system-users-information" );
    copy-of $result2;
}
```

Output:

```
show version:
Hostname: srx210
Model: srx210h
JUNOS Software Release [11.1R1.10]
show system users:
 3:01PM  up 12:02, 1 user, load averages: 0.24, 0.08, 0.02
USER     TTY      FROM                              LOGIN@  IDLE WHAT
jnpr     p1       10.0.0.50                        12:03PM     - -cli (cli)
```

Copy-of can also be used when an existing result tree fragment variable should be converted into a node-set through the node-set conversion operator. The **copy-of** statement copies the entire contents of the result tree fragment variable into the result tree where it is then redirected into the node-set conversion operator, which converts the contents into a node-set:

```
var $rtf = <command> "clear bgp neighbor";
var $ns := {
    copy-of $rtf;
}
```

Another use for the **copy-of** statement is to copy the desired XML data into the **<xsl:document>** instruction element so that it can be saved to a disk file:

Example code:

```
<xsl:document href="/var/tmp/output" indent="yes"> {
    copy-of $junos-context/user-context;
}
```

Output:

```
jnpr@srx210> file show /var/tmp/output
<?xml version="1.0"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
<user>jnpr</user>
<class-name>j-super-user-local</class-name>
<uid>2001</uid>
<login-name>jnpr</login-name>
</user-context>
```

Example

This op script demonstrates how the **copy-of** statement can be used to cause Junos to display its command results in their normal CLI format.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $command = jcs:get-input( "Enter CLI command to run: " );

        /* Run the command using the <command> RPC */
        var $command-rpc = <command> $command;
        var $results = jcs:invoke( $command-rpc );

        /* Copy the results to the result tree, this causes Junos to process
           the XML results and display them the same way they would if the
           command had been entered from the CLI */
        copy-of $results;
    }
}
```

Output

```
jnpr@srx210> op copy-of
Enter CLI command to run: show route 0.0.0.0/0 exact

inet.0: 7 destinations, 7 routes (7 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 12:16:06
                    > to 10.0.0.1 via ge-0/0/0.0

jnpr@srx210> op copy-of
Enter CLI command to run: show version
Hostname: srx210
Model: srx210h
JUNOS Software Release [11.1R1.10]
```

```
jnpr@srx210> op copy-of
Enter CLI command to run: invalid command
error: syntax error, expecting <command> or </command>: invalid
```

expr

Minimum Version: Junos 8.2

Syntax

expr *expression;*

Description

The **expr** statement evaluates its associated XPath expression, converts the result to a string, and writes it to the result tree. The XPath expression could consist of a function call, a location path, a literal number or string, etc., and SLAX-specific operators are permitted. The **expr** statement is most commonly used to invoke functions that return no results, for conditional variable assignment, and to return text content from a template. This statement cannot be used at the top-level of a script. It can only appear within a code block.

SLAX syntax rules force a function's results to be handled in some manner whether assigning them to a variable, writing them to the result tree, or using them within a larger expression. This proves awkward, however, with functions that never return a result, such as the **jcs:output**() function, because SLAX syntax does not allow the function to simply be called with no provision provided for its (nonexistent) results. The **expr** statement is a common solution for this scenario: where a function should be called that returns no results. The function is invoked by **expr** and nothing is written to the result tree because the function returns nothing to write:

```
expr jcs:output( "Display message" );
```

A second common use for the **expr** statement is performing conditional variable assignment. Because SLAX variables are immutable, it is necessary to assign their values carefully, and the **expr** statement provides a way to do this. It can write the expression's text content to the result tree, based on an **if** statement, and this text can then be redirected into a variable:

```
var $account-type = {
    if( $user == "jnpr" ) {
        expr "admin";
    }
    else {
        expr "other";
    }
}
```

As a result of this code, the $account-type variable contains either the text content "admin" or the text content "other". The conditional assignment structure creates a result tree fragment variable rather than an actual string variable, but SLAX automatically converts result tree fragments to strings when necessary, so it can be treated as if it were truly a string variable.

Another use for the **expr** statement is to return text content from a template. Technically, templates do not actually return any values but instead write their results into the result tree; however, if what is written is redirected into a variable, and the template only inserts text content, then for all intents and purposes the called template is returning a string to the calling template:

```
template to-lower( $string ) {
    expr translate( $string, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
        'abcdefghijklmnopqrstuvwxyz');
}
```

In the example above, the **expr** statement within the template invokes the **translate**() function and writes the string result to the result tree. This text content can then be redirected into a variable by the calling template in the following manner:

```
var $lower-case = { call to-lower( $string = "Junos" ); }
```

Similar to conditional variable assignment, this method causes the $lower-case variable to be a result tree fragment rather than an actual string variable; however, due to the automatic data type conversion performed by SLAX, the variable can be treated by the script as if it were a string.

Example

This op script demonstrates the three common uses of the **expr** statement.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Ask for the string and the action to perform */
        var $string = jcs:get-input( "Enter string to convert: " );
        var $action = jcs:get-input( "Convert to 'upper' or 'lower' case?: " );

        /* Modify the string according to the instructions */
        var $new-string = {
            if( $action == "upper" ) {
                /* Text content will be redirected into the variable */
                call to-upper( $string );
            }
            else if( $action == "lower" ) {
                /* Text content will be redirected into the variable */
                call to-lower( $string );
            }
            else {
                /* No action, just use the existing string value */
                expr $string;
            }
        }

        /* Display to the screen */
        expr jcs:output( $new-string );
    }
}

template to-lower( $string ) {
    expr translate( $string, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
        'abcdefghijklmnopqrstuvwxyz');
}
```

```
template to-upper( $string ) {
    expr translate( $string, 'abcdefghijklmnopqrstuvwxyz',
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
}
```

Output

```
jnpr@srx210> op expr
Enter string to convert: SLAX 1.0 Reference Guide
Convert to 'upper' or 'lower' case?: lower
slax 1.0 reference guide

jnpr@srx210> op expr
Enter string to convert: SLAX 1.0 Reference Guide
Convert to 'upper' or 'lower' case?: upper
SLAX 1.0 REFERENCE GUIDE
```

for-each

Minimum Version: Junos 8.2

Syntax

for-each*(node-set-expression) {*
...loop code...
}

Description

The **for-each** statement is used to loop through a code block for each node within a selected node list. The statement includes a node-set expression, contained within parentheses, followed by a code-block, contained within curly brackets. The node-set XPath expression is first evaluated, resulting in a node-set that becomes the current node list for the code block. The code block is then repeated with each node in the node list serving as the current node for one iteration.

Code example:

```
var $configuration = jcs:invoke( "get-configuration" );
for-each( $configuration/interfaces/interface ) {
    <output> name;
}
```

Output:

```
ge-0/0/0
ge-0/0/1
fe-0/0/2
ge-0/0/2
xe-0/1/0
lo0
```

By default, the current node list is arranged with the nodes in document order; however, **<xsl:sort>** instruction elements can be added at the beginning of the **for-each** code block to instruct the script processor to sort the node list in a different manner. The prior code example displayed the interfaces in document order, which is the

same order as they appear within the configuration. This example uses **<xsl:sort>** to order them alphabetically instead:

Code example:

```
var $configuration = jcs:invoke( "get-configuration" );
for-each( $configuration/interfaces/interface ) {
    <xsl:sort select="name">;
    <output> name;
}
```

Output:

```
fe-0/0/2
ge-0/0/0
ge-0/0/1
ge-0/0/2
lo0
xe-0/1/0
```

Unlike a traditional for loop, a SLAX **for-each** loop does not provide a way to simply loop through a code block a given number of times; instead, the **for-each** statement always processes its code block once for every node in its selected node-list, so the number of iterations depends on the size of the node list selected by the **for-each** statement's node-set expression. Also, there is no way to break out of a SLAX **for-each** loop early. It will always iterate through each member of its node list.

The **current**(), **last**(), and **position**() functions provide access to the current node, the size of the current node list, as well as the current node's position within the current node list. Misuse of the **position**() function can result in errors if a script includes a **for-each** loop that selects a subset of nodes within a node-set and the **position**() function is used with the misunderstanding that the position is based on the index of the node within the original node-set, which it is not; rather, its reported position is its index within the current node list, selected by the node-set expression of the **for-each** loop. To illustrate, this example contains two **for-each** statements, one which displays all of the interfaces present in the configuration , and the other that only shows a subset:

Code example:

```
var $configuration = jcs:invoke( "get-configuration" );
/* All interfaces */
for-each( $configuration/interfaces/interface ) {
    <output> position() _ " of " _ last() _ " = " _ name;
}
/* GE only */
for-each( $configuration/interfaces/interface[ starts-with( name, "ge" )]) {
    <output> position() _ " of " _ last() _ " = " _ name;
}
```

Output:

```
1 of 6 = ge-0/0/0
2 of 6 = ge-0/0/1
3 of 6 = fe-0/0/2
4 of 6 = ge-0/0/2
5 of 6 = xe-0/1/0
6 of 6 = lo0
1 of 3 = ge-0/0/0
2 of 3 = ge-0/0/1
3 of 3 = ge-0/0/2
```

The first **for-each** loop iterates through all of the interface nodes in document order, which results in the interface ge-0/0/2 being processed at position 4 of 6; however, the second **for-each** loop only iterates through a subset of the interface nodes, so interface ge-0/0/2 is now at position 3, which is different than its actual

document position of 4.

This statement cannot be used at the top-level of a script. It can only appear within a code block.

Example

This op script demonstrates how to use the **for-each** statement to loop through the file systems and display them in a particular order.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve storage information */
        var $storage = jcs:invoke( "get-system-storage" );

        /* Display the filesystems, sorted based on their size */
        <output> "Filesystems, based on size:";
        <output> jcs:printf( "%-15s %10s", "Name", "Size" );
        for-each( $storage/filesystem ) {
            /* Primary sort is the size */
            <xsl:sort select="total-blocks" data-type="number" order="descending">;
            /* Secondary sort is the name */
            <xsl:sort select="filesystem-name">;
            <output> jcs:printf( "%-15s %10s", filesystem-name, total-blocks/@junos:format );
        }
    }
}
```

Output

```
Filesystems, based on size:
Name                  Size
/dev/md0              360M
/cf/var/jail          342M
/cf/var/log           342M
/dev/bo0s3f           342M
/cf                   293M
/dev/da0s2a           293M
/dev/md1              168M
/dev/md2               39M
/dev/bo0s3e            24M
/dev/md3              1.8M
procfs                4.0K
devfs                 1.0K
devfs                 1.0K
devfs                 1.0K
```

if

Minimum Version: Junos 8.2

Syntax

if*(boolean-expression) {*
...conditional code...
}
else if*(boolean-expression) { /* optional – can appear multiple times */*
...conditional code...
}
else *{ /* optional */*
...conditional code...
}

Description

The **if** statement is used to conditionally execute a code block, based on the result of its boolean XPath expression that is enclosed within parentheses:

```
if( $input == "yes" ) {
    call display-results();
}
```

In the example above, the boolean expression is: $input == “yes”. If it evaluates to true, then the code block, which must be enclosed within curly brackets, is executed, causing the display-results template to be called, but if the boolean expression evaluates to false then the code block is not executed, and script processing commences at the line following the end of the code block.

An **else** code block can be included for the **if** statement, providing an alternate path to follow if the **if** statement’s boolean expression is false:

```
var $day-name = date:day-name();
if( $day-name == "Saturday" || $day-name == "Sunday" ) {
    <output> "It is the weekend!";
}
else {
    <output> "Back to work!";
}
```

When the **if** statement’s boolean expression evaluates to true, the output string in the **if** code block is displayed, but when the expression evaluates to false, the output string in the **else** code block is displayed instead. In all cases, however, only one code block or the other is executed.

In addition to **else**, an **if** statement can be followed by one or more **else if** code blocks. These code blocks follow the **if** statement but precede the **else** code block (if one exists). They contain a boolean expression, like the **if** statement, that is enclosed in parentheses, but these boolean expressions are only evaluated if the **if** statement’s expression, and all earlier **else if** expressions connected to that **if** statement, are false. In other words, only one code block is ever executed, and it is always the first code block with a boolean expression that is true, or if all the **if** and **else if** boolean expressions are false then the **else** code block is executed (if one exists).

```
var $rpc-string = {
    if( $input == "show route" ) {
        expr "get-route-information";
    }
```

```
    else if( $input == "show version" ) {
        expr "get-software-information";
    }
    else if( $input == "show chassis hardware" ) {
        expr "get-chassis-inventory";
    }
    else {
        expr "Unknown";
    }
}
```

The above example demonstrates conditional variable assignment. The $rpc-string is only assigned one of the four possible text strings. If the **if** or any of the **else if** boolean expressions evaluate to true then that value will be assigned to $rpc-string; otherwise, if they are all false then the **else** code block is executed and the $rpc-string is assigned the value of "Unknown".

This statement cannot be used at the top-level of a script. It can only appear within a code block.

Example

This op script demonstrates how to use the **if** statement.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display the script type */
        if( $junos-context/script-type == "commit" ) {
            <output> "This is a commit script";
        }
        else if( $junos-context/script-type == "event" ) {
            <output> "This is an event script";
        }
        /* op script */
        else {
            /* Was it executed via op url */
            if( $junos-context/op-context/via-url ) {
                <output> "This is an op script executed via a url";
            }
            else {
                <output> "This is an op script";
            }
        }
    }
}
```

Output

```
jnpr@srx210> op if
This is an op script
```

```
jnpr@srx210> op url file://var/db/scripts/op/if.slax
This is an op script executed via a url
```

import

Minimum Version: Junos 8.2

Syntax

import *"import-script";*

Description

The **import** statement is used to import code from a separate import file into the executing script. The imported code is always treated as a lower precedence than the code within the importing script, so any duplicate items that are contained within the import file are overridden by their counterparts in the importing script. The junos.xsl import file is imported as part of the standard script boilerplate. It should always be imported rather than included in order to ensure that a "match /" template within the importing script will never be overridden by the "match /" template found within the junos.xsl import file.

The **import** statement must appear immediately following any namespace declarations and before any other statements or instructions. Multiple files can be imported, and the latter ones have precedence over the prior ones, but the importing script has precedence over them all. The name of the import script is provided within quotes following the **import** statement. Imported scripts can be written in either XSLT or SLAX, and the filename can be provided with either an absolute or relative path. If a relative path is used then it is relative to the script file's location, which will be either the commit, event, or op subdirectory of the main script directory (Except for scripts executed via "op url" as explained below).

In Junos 11.1, a new subdirectory was added to the main script directory for the purpose of storing third-party libraries: lib. It is recommend that all import files be placed within that directory to provide similar access to the file whether it is being imported by a commit script, an event script, or an op script.

```
import "../lib/third-party.slax";
```

When op scripts are run remotely via the "op url" command, they are copied to a temporary local directory and then executed locally, so any import files must be present on the local file system rather than on the remote server. A symbolic link is present within the temporary directory, pointing to the import script directory, so the boilerplate import line that imports the junos.xsl script via "../import/junos.xsl" will work as expected; however, at the time of this writing, the temporary directory contains no symbolic link to the lib directory, or any of the other script directories, so any third-party import files must be specified with absolute paths rather than relative ones.

```
import "/var/run/scripts/lib/third-party.slax";
```

Example

This op script demonstrates how to import a library using the **import** statement.

Code

```
version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns example = "http://xml.juniper.net/example";

/* junos.xsl import file should always be imported, not included */
import "../import/junos.xsl";

/* lib directory was added in Junos 11.1 */
import "../lib/third-party.slax";

match / {
    <op-script-results> {

        var $input = jcs:get-input( "Enter a string: " );
        <output> "In uppercase: " _ example:to-upper( $input );
        <output> "In lowercase: " _ example:to-lower( $input );
    }
}
```

third-party.slax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns func extension = "http://exslt.org/functions";
ns example = "http://xml.juniper.net/example";

/* Translate a string into lower case */
<func:function name="example:to-lower"> {
    param $string;
    <func:result select=
        "translate($string,'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')">;
}

/* Translate a string into upper case */
<func:function name="example:to-upper"> {
    param $string;
    <func:result select=
        "translate($string,'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')">;
}
```

Output

```
jnpr@srx210> op import
Enter a string: Junos SLAX Script
In uppercase: JUNOS SLAX SCRIPT
In lowercase: junos slax script
```

include

Minimum Version: Junos 8.2

Syntax

include *"import-script";*

Description

The **include** statement is used to include code from a separate file into the script during processing. Unlike the **import** statement, when the **include** statement is used the included code is treated as if it were part of the actual script file, meaning that any script components that are not allowed to be redefined, such as global variables and named templates, cannot be present in both the script and its include files. Because of this behavior, it is more common to import script files rather than include them. In particular, the junos.xsl import file should always be imported rather than included to ensure that the "match /" template in the script file is not overridden by the "match /" template in the junos.xsl import file.

The **include** statement can be placed anywhere in the top-level of the script so long as it is after any **import** statements and namespace definitions. The name of the script is provided within quotes following the **include** statement. The **include** statement can be used multiple times, allowing more than one script to be included. Included scripts can be written in either XSLT or SLAX, and the filename can be provided with either an absolute or relative path. If a relative path is used then it is relative to the script file's location, which will be either the commit, event, or op subdirectory of the main script directory (Except for scripts executed via "op url" as explained below).

In Junos 11.1, a new subdirectory was added to the main script directory for the purpose of storing third-party libraries: lib. It is recommend that all include files be placed within that directory to provide similar access to the file whether it is being included by a commit script, an event script, or an op script.

```
include "../lib/third-party.slax";
```

When op scripts are run remotely via the "op url" command, they are copied to a temporary local directory and then executed locally, so any include files must be present on the local filesystem rather than on the remote server. At the time of this writing, the temporary directory contains no symbolic link to the lib directory, so any include files must be specified with absolute paths rather than relative ones.

```
include "/var/run/scripts/lib/third-party.slax";
```

Example

This op script demonstrates how to include a library using the **include** statement.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns example = "http://xml.juniper.net/example";

/* junos.xsl import file should always be imported, not included */
import "../import/junos.xsl";

/* lib directory was added in Junos 11.1 */
include "../lib/third-party.slax";

match / {
    <op-script-results> {

        var $input = jcs:get-input( "Enter a string: " );
        <output> "In uppercase: " _ example:to-upper( $input );
        <output> "In lowercase: " _ example:to-lower( $input );
    }
}
```

third-party.slax

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns func extension = "http://exslt.org/functions";
ns example = "http://xml.juniper.net/example";

/* Translate a string into lower case */
<func:function name="example:to-lower"> {
    param $string;
    <func:result select=
        "translate($string,'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')">;
}

/* Translate a string into upper case */
<func:function name="example:to-upper"> {
    param $string;
    <func:result select=
        "translate($string,'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')">;
}
```

Output

```
jnpr@srx210> op include
Enter a string: Juniper Networks
In uppercase: JUNIPER NETWORKS
In lowercase: juniper networks
```

match

Minimum Version: Junos 8.2

Syntax

match *pattern {*
...optional mode/priority...
...optional parameters...
...template code...
}

Description

The **match** statement is used at the top-level of a script to create a match template. The statement is followed by the template's match pattern and then its code block, enclosed within curly brackets:

```
match interfaces {
    expr jcs:output( "This is an interfaces node" );
}
```

Match patterns indicate what nodes should match a particular match template. They are similar to location paths except they are evaluated without a specific context. If a node matches the pattern with the context rooted at itself or any of its ancestors then it is considered to be a match. For example, a template with a match pattern of "name" would match a /interfaces/interface/name node, as well as an /interfaces/interface/unit/name

node, and it would match a /firewall/filter/name node as well.

The boilerplate of all script types includes a match template, which is the starting point of the script instructions. For op scripts and event scripts, this is the "match /" template:

```
match / {
    ...script code...
}
```

When the script processor starts executing the script, it searches for a matching template for the source tree's root node. Upon finding the "match /" template in the script file the script processor commences to process its instructions.

The starting process for commit scripts is a little different because they do not have a "match /" template within their boilerplate; instead, the "match /" template imported from the junos.xsl file takes effect first:

```
<xsl:template match="/">
    <commit-script-results>
        <xsl:apply-templates select="commit-script-input/configuration"/>
    </commit-script-results>
</xsl:template>
```

This "match /" template, expressed in XSLT in the junos.xsl import file, uses the **apply-templates** statement to select a match template for the source tree's <configuration> node, and encloses the results of that template within the <commit-script-results> result tree element. The script processor identifies the "match configuration" template, which is part of the commit script boilerplate, as the best match for the <configuration> node and begins to process the script's instructions starting at that point.

Match templates can include **mode** and **priority** statements at the beginning of their code block if a template mode or template priority should be set. Refer to the descriptions of these SLAX statements for details on their use and the effect they have on the template processing.

Parameters can be defined for a match template by including the **param** statement at the beginning of the code block, following the **mode** and **priority** statements (if they exist):

```
match rt-entry {
    param $display;

    <output> "Route: " _ ../rt-destination;
    if( $display == "next-hop" ) {
        <output> "Next-hop = " _ nh/to;
    }
    else {
        <output> "Protocol = " _ protocol-name;
    }
}
```

Unlike with custom functions, the order of the parameter definitions is insignificant. Parameter values are always passed to templates by name, not by position.

Match templates are invoked through the **apply-templates** statement. The script processor iterates through each node selected by **apply-templates**, finding the best match template for each node, and invokes that match template with the selected node as the current node.

The current node and current node list are changed by the **apply-templates** statement, causing them to be different within the match template than they are within the calling template. This behavior is in contrast to named templates where the current node and current node list remain the same inside the called template as they are inside the calling template.

Match templates do not have a return value like custom functions do; instead, they write content to the result

tree, which can be redirected by the calling function into a variable if desired. See the description of the **apply-templates** statement for details.

Other than the starting boilerplate template, match templates are not commonly used within Junos SLAX scripts; instead, most programmers modularize their code with named templates or custom functions.

Example

This op script demonstrates how to use a match template to display the configured address for all interfaces.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $configuration = jcs:invoke( "get-configuration" );

        /* iterate through all of the addresses */
        apply-templates $configuration/interfaces/interface/unit/family/inet/address;

    }
}

match family/inet/address {
    /* Display the interface address */
    <output> ../../../../name _ "." _ ../../../name _ " address = " _ name;
}
```

Output

```
ge-0/0/0.0 address = 10.0.0.10/24
ge-0/0/1.0 address = 10.100.100.1/24
fe-0/0/2.0 address = 10.50.50.1/24
ge-0/0/2.0 address = 10.100.100.1/24
xe-0/1/0.0 address = 10.200.200.1/24
lo0.0 address = 127.0.0.1/32
```

mode

Minimum Version: Junos 8.2

Syntax

mode *“string”*;

Description

Template modes provide an additional way to indicate which match template should be selected for a node through the **apply-templates** statement. The **mode** statement includes a string value, which must be a valid qualified name (see Appendix B), identifying the mode that should be used. (This string must be hardcoded; it cannot be set through a variable). When a mode is specified as part of the **apply-templates** statement, only match templates that have the same mode configured are considered as a possible template to process the node with; likewise, if no mode is specified by the **apply-templates** statement then only match templates with no mode configured are considered.

Code example:

```
match / {
    <op-script-results> {
        var $string := {
            <string> "Junos";
        }
        /* Call with no mode */
        apply-templates $string/string;
        /* Call with upper-case mode */
        apply-templates $string/string {
            mode "upper-case";
        }
        /* Call with lower-case mode */
        apply-templates $string/string {
            mode "lower-case";
        }
    }
}
/* No mode */
match string {
    <output> .;
}
/* upper-case mode */
match string {
    mode "upper-case";
    <output> translate( ., 'abcdefghijklmnopqrstuvwxyz',
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
}
/* lower-case mode */
match string {
    mode "lower-case";
    <output> translate( ., 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
        'abcdefghijklmnopqrstuvwxyz');
}
```

Output:

```
Junos
JUNOS
junos
```

A mode is configured by including the **mode** statement in the code block of an **apply-templates** statement or a match template. When included in a match template's code block, it must appear before all other code except for the **priority** statement.

Example

This op script demonstrates how to use the **mode** statement to differentiate between two match templates that have the same match pattern.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve the user account configuration */
        var $user-rpc = {
            <get-configuration database="committed" inherit="inherit"> {
                <configuration> {
                    <system> {
                        <login>;
                    }
                }
            }
        }
        var $users = jcs:invoke( $user-rpc );

        /* Decide whether to display the UIDs or the classes */
        var $input = jcs:get-input( "Display 'uids' or 'classes'? " );

        if( $input == "uids" ) {
            apply-templates $users/system/login/user {
                mode "uids";
            }
        }
        else if( $input == "classes" ) {
            apply-templates $users/system/login/user {
                mode "classes";
            }
        }
        else {
            <output> "Invalid selection";
        }

    }
}

/* uids mode */
match user {
    mode "uids";

    /* Display user and their UID */
    <output> "User: " _ name _ " UID: " _ uid;
}

/* classes mode */
match user {
    mode "classes";

    /* Display user and their UID */
    <output> "User: " _ name _ " Class: " _ class;
}
```

Output

```
jnpr@srx210> op mode
Display 'uids' or 'classes'? uids
User: jnpr UID: 2001
User: new-admin UID: 2004
User: test UID: 2006
User: user3 UID: 2005
User: operator-local UID: 2000

jnpr@srx210> op mode
Display 'uids' or 'classes'? classes
User: jnpr Class: super-user-local
User: new-admin Class: super-user
User: test Class: read-only
User: user3 Class: super-user
User: operator-local Class: operator
```

ns

Minimum Version: Junos 8.2

Syntax

ns *prefix1 = "namespace-uri1";*
ns *prefix2* **extension** *= "namespace-uri2";*
ns *prefix3* **exclude** *= "namespace-uri3";*
ns *"default-namespace-uri";*

Description

The **ns** statement defines a namespace, which consists of a prefix and a namespace URI. The prefix can then be used as part of a node's name to indicate the node's assigned namespace URI. In almost all cases, namespaces are defined at the beginning of the script immediately after the **version** statement and before any other statements or instructions. This placement causes the namespace to be defined throughout the script. But, the **ns** statement can be used within the script itself at the beginning of any code block if the namespace should only be active within that section of the code; however, this is not a common coding practice in Junos SLAX scripts.

In many cases, the only namespaces that are defined in a Junos SLAX script are the three boilerplate namespaces:

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

Other namespaces are usually only added in order to use a function that requires a namespace, such as those that come from EXSLT, or if the script uses custom functions, which must include a namespace prefix as part of their name. When creating a custom namespace, the URI should be something that the script writer, or their organization, has ownership of.

Namespaces can be declared as extension namespaces by including the **extension** keyword after the prefix. This instructs the script processor to treat all elements with that namespace within the script's code as extension elements, meaning that they will be processed as instructions rather than data. The most common

namespace that is defined as an extension namespace is the EXSLT Functions namespace, which typically uses the prefix: “func”.

```
ns func extension = "http://exslt.org/functions";
```

Namespaces can also be declared as excluded namespaces by including the **exclude** keyword after the prefix. This is intended to exclude the namespace prefix from the result tree; however, the Junos script processor strips namespaces from the result tree, so there is little need to use this option with Junos SLAX scripts.

A default namespace can be defined by not including a prefix for the namespace and only including the **ns** statement with a following namespace URI string. This is not a common coding practice for Junos SLAX scripts and is unlikely to be necessary in most cases.

Example

This op script demonstrates different namespace definitions that might be used within a SLAX script.

Code

```
version 1.0;

/* The three boilerplate namespace definitions */
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

/* Define the EXSLT Strings namespace so that the str:padding() function can
   be used */
ns str = "http://exslt.org/strings";

/* Define the EXSLT Functions namespace as an extension so that <func:result>
   will be processed correctly */
ns func extension = "http://exslt.org/functions";

/* Define a custom namespace to use for the custom function name */
ns example = "http://xsl.juniper.net/example";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get a string, padding, and length */
        var $string = jcs:get-input( "Enter a string: ");
        var $padding = jcs:get-input( "Enter a padding character: " );
        var $length = jcs:get-input( "How much padding?: " );

        /* Call the custom function */
        <output> example:surround-string( $string, $padding, $length );
    }
}

/* Custom function */
<func:function name="example:surround-string"> {
    param $string;
    param $padding;
    param $length;

    /* Return the string with the padding at the beginning and end */
    var $padding-string = str:padding( $length, substring( $padding, 1, 1) );
```

```
    <func:result select="concat( $padding-string, $string, $padding-string )">;
}
```

Output

```
jnpr@srx210> op ns
Enter a string: My Example
Enter a padding character: :
How much padding?: 5
:::::My Example:::::
```

param

Minimum Version: Junos 8.2

Syntax

param *$name;*
param *$name = default-value;*

Description

The **param** statement creates a parameter, which is similar to a variable except that its value is provided through an outside source. Following the **param** statement, the parameter's name is given, which must begin with a dollar sign "$" and must be a valid qualified name (see Appendix B). A default value can optionally be assigned, which is used as the parameter's value if it is not otherwise set. If no default value is specified, and the parameter is not given a value when the script is executed, then the parameter defaults to a blank string. Parameters can be any of the standard data types, but the type is set automatically based on their value assignment. (Quoted values become strings, unquoted numeric values become numbers, etc.). At the time of this writing, parameters that are set through command-line arguments are always strings. Like variables, parameters are immutable. Once their value has been set, it cannot be changed.

Parameters that are defined at the top-level of the script are known as global parameters, and they can be referenced from within any of the script's templates or custom functions. The junos.xsl import file, which is imported when the standard script boilerplate is used, contains a number of default global parameters that are available to all script types including $hostname, $localtime, and $user. Additional global parameters can be created by defining them through the **param** statement at the top-level of the script. The value of these parameters is provided based on their script type:

- Op script global parameters are given values through command-line arguments.
- Event script global parameters are given values through the event policy configuration.
- Commit scripts have no way to provide values to custom global parameters.

```
param $remote-host;
param $user-name = "admin";
```

Template parameters can be created by defining a parameter using the **param** statement within a match or named template. Multiple **param** statements can be included, creating multiple parameters, but no other template code is allowed prior to the **param** statements (except for the **mode** and **priority** statements in a match template). When the template is called, the parameters can be provided values; otherwise, the parameters are assigned their default values when the template executes.

```
template build-loop() {
    param $index = 1;
    param $maximum;
    ...
}
```

The **param** statement is also used within custom functions that are created through the **<func:function>** extension element, but in this case they signify the function's arguments. The same syntax is followed as a global or template parameter, and the same opportunity exists to provide a default value. Also, just like with template parameters, all **param** statements within custom functions must be coded prior to any other function code.

```
<func:function name="example:to-lower"> {
    param $string;
    <func:result select=
"translate($string,'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')">;
}
```

Example

This op script demonstrates the use of both global parameters, which in op scripts are command-line arguments, as well as template parameters.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Define the argument description for CLI help */
var $arguments = {
    <argument> {
        <name> "string";
        <description> "String that should be split";
    }
    <argument> {
        <name> "position";
        <description> "Position where split should occur";
    }
}

/* Global Parameters: */
param $string;
/* Default provided to $position argument */
param $position = string-length( $string ) div 2;

match / {
    <op-script-results> {

        /* Verify input arguments */
        if( string-length( $string ) <= $position ) {
            <xsl:message terminate="yes"> "Invalid string and/or split position";
        }

        var $first = substring( $string, 1, $position );
        var $second = substring( $string, $position + 1 );
        call display-parts( $first, $second );
```

```
    }
}

template display-parts() {
    /* Template parameters:
       These could have been initialized within the parenthesis instead
       as ( $first, $second ) */
    param $first;
    param $second;

    <output> "First part: " _ $first;
    <output> "Second part: " _ $second;
}
```

Output

```
jnpr@srx210> op param string 1234567890 position 4
First part: 1234
Second part: 567890
```

preserve-space

Minimum Version: Junos 8.2

Syntax

preserve-space *element;*
preserve-space **;*

Description

The **preserve-space** statement instructs the script processor to preserve any whitespace-only child text nodes from the source tree element node listed. A text node is considered to only contain whitespace if it has no characters other than space, tab, newline, and carriage return. Preserving whitespace-only child text nodes is the default behavior of the script processor, so the **preserve-space** statement is only needed if the **strip-space** statement has been used with an asterisk, indicating that whitespace-only child text nodes should be removed from all element nodes. In this scenario, the **preserve-space** statement might be included to indicate specific element nodes that should not have their whitespace-only child text nodes stripped.

The **preserve-space** statement must appear at the top-level of the script. Multiple **preserve-space** statements can be included, with one element specified per statement. The asterisk can alternatively be specified, indicating that all elements should be preserved from whitespace stripping, but that is the default behavior any way. Junos does not support the third form: namespace-prefix:*, so the value of the **preserve-space** statement must be either an element name or the asterisk.

Whitespace stripping is only performed on the source tree and on XML documents read through the **document**() function. It is not performed on the XML documents generated by the Junos **jcs:invoke**() or **jcs:execute**() functions.

Example

This op script demonstrates how to use the **preserve-space** statement to counteract the **strip-space** statement

and preserve the whitespace on one element node.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Remove the whitespace text nodes from all elements except for <user-context> */
preserve-space user-context;
strip-space *;

match / {
    <op-script-results> {
        /* Write to disk without indentation */
        <xsl:document href="/var/tmp/op-script-source-tree"> {
            copy-of /op-script-input;
        }
    }
}
```

Output

```
jnpr@srx210> op url file://var/db/scripts/op/preserve-space.slax

jnpr@srx210> file show /var/tmp/op-script-source-tree
<?xml version="1.0"?>
<op-script-input xmlns:junos="http://xml.juniper.net/junos/*/junos"><junos-
context><hostname>srx210</hostname><product>srx210h</product><localtime>Sat May  7 04:51:27
2011</localtime><localtime-iso>2011-05-07 04:51:27 UTC</localtime-iso><script-type>op</script-
type><pid>2409</pid><tty>/dev/ttyp0</tty><chassis>others</chassis><routing-engine-name>re0</
routing-engine-name><re-master/><user-context>
<user>jnpr</user>
<class-name>j-super-user-local</class-name>
<uid>2001</uid>
<login-name>jnpr</login-name>
</user-context><op-context><via-url/></op-context></junos-context></op-script-input>
```

priority

Minimum Version: Junos 8.2

Syntax

priority *value;*

Description

The **priority** statement is used within a match template to define the template's priority. When a node matches more than one match template with the same import precedence then the template with the highest priority is selected, or, if both have the same priority, then the one located later in the script is used.

The **priority** statement must appear within the match template's code block before all other code, including **param** statements, with the exception of the **mode** statement. The priority value can be either a positive or negative number and can include a decimal component. If no priority is specified then the default priority is assigned to the match template.

This table identifies the default priorities for match templates, based on the style of their pattern. If the pattern contains the node-set union operator then each part of the pattern is evaluated separately:

Match pattern	Example	Priority
Node test on the child axis (the default) or on the attribute axis	match node()	-0.5
Wildcard namespace prefix match on the child axis (the default) or on the attribute axis	match xnm:*	-0.25
Element name on the child axis (the default) or on the attribute axis	match interfaces	0.0
Multiple steps and more complex patterns	match family/inet/address	0.5

At the time of this writing, manually specifying the child axis along with an element name in the match pattern: "match child::interfaces" causes the default priority to be 0.5, instead of 0.0, as it should be.

Example

This op script demonstrates how to use the **priority** statement to override the default template priorities and cause a different template to be selected than would have otherwise been chosen.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $configuration = jcs:invoke( "get-configuration" );

        /* iterate through all of the addresses */
        apply-templates $configuration/interfaces/interface/unit/family/inet/address;

    }
}

/* Default priority is 0.0 */
match address {
    /* Adjust the priority so that it takes precedence */
    priority 0.6;
    <output> ../../../../name _ "." _ ../../../name _ " address = " _ name;
}

/* Default priority is 0.5 */
```

```
match family/inet/address {
    <output> "Address: " _ name;
}
```

Output

```
ge-0/0/0.0 address = 10.0.0.10/24
ge-0/0/1.0 address = 10.100.100.1/24
fe-0/0/2.0 address = 10.50.50.1/24
ge-0/0/2.0 address = 10.100.100.1/24
xe-0/1/0.0 address = 10.200.200.1/24
lo0.0 address = 127.0.0.1/32
```

strip-space

Minimum Version: Junos 8.2

Syntax

strip-space *element;*
strip-space **;*

Description

The **strip-space** statement instructs the script processor to remove any whitespace-only child text nodes from the source tree element node listed. A text node is considered to only contain whitespace if it has no characters other than space, tab, newline, and carriage return. The **strip-space** statement must appear at the top-level of the script. Multiple **strip-space** statements can be included, with one element specified per statement. The asterisk can alternatively be specified, indicating that all elements should have their whitespace-only child text nodes stripped; however, at the time of this writing, Junos does not support the third form: namespace-prefix:*, so the value of the **strip-space** statement must be either an element name or the asterisk.

By default, all whitespace-only child text nodes are preserved, so whitespace stripping is only performed on the child text nodes of the elements indicated by **strip-space** statements, but it is not performed on the child text nodes of their element node children; instead, those element nodes must also be designated for whitespace stripping via a **strip-space** statement, or the asterisk must be specified, indicating that all element nodes should have their whitespace-only child text nodes removed.

Even with the **strip-space** statement included, whitespace stripping is only performed on the source tree and on XML documents read through the **document**() function. It is not performed on the XML documents generated by the Junos **jcs:invoke**() or **jcs:execute**() functions; however, if whitespace-only text nodes need to be removed from those results then one workaround would be to write the results to disk using the **<xsl:document>** instruction element and then to read them using the **document**() function, which is impacted by the **strip-space** statements.

Junos inserts a newline between every node in the source tree by default:

```
<op-script-input xmlns:junos="http://xml.juniper.net/junos/*/junos">
<junos-context>
<hostname>srx210</hostname>
<product>srx210h</product>
<localtime>Sat May  7 03:42:44 2011</localtime>
<localtime-iso>2011-05-07 03:42:44 UTC</localtime-iso>
```

```
<script-type>op</script-type>
<pid>2118</pid>
<tty>/dev/ttyp0</tty>
<chassis>others</chassis>
<routing-engine-name>re0</routing-engine-name>
<re-master/>
<user-context>
<user>jnpr</user>
<class-name>j-super-user-local</class-name>
<uid>2001</uid>
<login-name>jnpr</login-name>
</user-context>
<op-context>
<via-url/>
</op-context>
</junos-context>
</op-script-input>
```

Generally, this has no impact on scripts, as the relevant text content does not have any extra newlines added; however, it does prevent the source tree from being properly indented when writing it to an output file via **<xsl:document>**. The **strip-space** statement provides a solution to this, because it can be used to remove all the extra whitespace text nodes, allowing the desired indentation to take place. The example script for this section demonstrates how this can be done.

Example

This op script demonstrates how to use the **strip-space** statement to remove all whitespace-only text nodes from the source tree, which allows the **<xsl:document>** instruction element to indent the XML document correctly when it is written to disk.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Remove all whitespace-only text nodes from the source tree */
strip-space *;

match / {
    <op-script-results> {
        /* Write to disk with proper indentation */
        <xsl:document href="/var/tmp/op-script-source-tree" indent="yes"> {
            copy-of /op-script-input;
        }
    }
}
```

Output

```
jnpr@srx210> op url file://var/db/scripts/op/strip-space.slax

jnpr@srx210> file show /var/tmp/op-script-source-tree
<?xml version="1.0"?>
<op-script-input xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <junos-context>
```

```
      <hostname>srx210</hostname>
      <product>srx210h</product>
      <localtime>Sat May  7 03:54:33 2011</localtime>
      <localtime-iso>2011-05-07 03:54:33 UTC</localtime-iso>
      <script-type>op</script-type>
      <pid>2166</pid>
      <tty>/dev/ttyp0</tty>
      <chassis>others</chassis>
      <routing-engine-name>re0</routing-engine-name>
      <re-master/>
      <user-context>
        <user>jnpr</user>
        <class-name>j-super-user-local</class-name>
        <uid>2001</uid>
        <login-name>jnpr</login-name>
      </user-context>
      <op-context>
        <via-url/>
      </op-context>
    </junos-context>
  </op-script-input>
```

template

Minimum Version: Junos 8.2

Syntax

template *name(...optional parameters...) {*
... template code...
}
template *name {*
...optional parameters...
.. template code...
}

Description

The **template** statement is used at the top-level of a script to create a named template. The template statement is followed by the template's name, which must be a valid qualified name (see Appendix B), and can optionally be followed by a set of parentheses. After the parentheses, or the template's name if no parentheses are present, the template's code block is enclosed within curly brackets:

```
template example {
    expr jcs:output( "This is an example" );
}
```

Including parentheses after the template name provides a shortened way to define template parameters: they are defined by listing the parameter names within the parentheses, separated by commas. Default value assignments can be included in this short form as well. When defining parameters within the named template parentheses, the **param** statement is not necessary:

```
template build-string( $index = 1, $length, $character ) {
    if( $index <= $length ) {
```

```
            expr $character;
        }
        if( $index < $length ) {
            call build-string( $index = $index + 1, $length, $character );
        }
    }
```

In the above example, the $index, $length, and $character parameters are defined for the template by including them within parentheses between the template's name and its code block. Only the $index parameter is given a default value.

The more verbose method to define parameters in a named template is to include the **param** statement at the beginning of the code block. The below template behaves the same as the one above, but its parameters are now defined through the **param** statement:

```
template build-string {
    param $index = 1;
    param $length;
    param $character;
    if( $index <= $length ) {
        expr $character;
    }
    if( $index < $length ) {
        call build-string( $index = $index + 1, $length, $character );
    }
}
```

Unlike with custom functions, the order of the parameter definitions is insignificant. Parameter values are always passed to templates by name, not by position.

Named templates are called with the **call** statement, as demonstrated in the two examples above, which call the build-string template recursively until the $index parameter matches the $length parameter.

A template cannot have the same name as another template within the same script file. Duplicate template names are only allowed if the templates have a different import precedence, so the same template name could appear within an import file, imported through the **import** statement, because the imported template would always have a worse import precedence than the template defined within the importing script file.

The current node and current node list are not changed when a named template is invoked. They remain the same as they are in the calling template. This behavior is in contrast to match templates, where the current node and current node list are changed by the **apply-templates** statement as it invokes the matching match templates.

Named templates do not have a return value like custom functions do; instead, they write content to the result tree, which can be redirected by the calling function into a variable if desired. See the description of the **call** statement for details.

Example

This op script demonstrates the different formats in which named templates can be created with the **template** statement.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $string = jcs:get-input( "Enter string: " );

        /* Display normally and in all caps and all lowercase */
        <output> $string;

        /* Call template with the parameters specified inside parenthesis */
        <output> {
            call to-upper( $string );
        }

        /* Call template with parameter specified via the with statement */
        <output> {
            call to-lower {
                with $string;
            }
        }

    }
}

/* Parameter defined within parentheses and given a default value */
template to-lower( $string = "no string" ) {
    expr translate( $string, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
        'abcdefghijklmnopqrstuvwxyz');
}

/* Parameter defined within the template's code block */
template to-upper {
    param $string;
    expr translate( $string, 'abcdefghijklmnopqrstuvwxyz',
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
}
```

Output

```
jnpr@srx210> op template
Enter string: Juniper Networks
Juniper Networks
JUNIPER NETWORKS
juniper networks
```

var

Minimum Version: Junos 8.2

Syntax

var *$name = value;*
var *$name = {*
  ... value...

}

Description

The **var** statement creates a variable and assigns its value. Following the **var** statement, the variable's name is given, which must begin with a dollar sign "$" and must be a valid qualified name (see Appendix B). A value should then be given, assigned with either the assignment operator or node-set conversion operator, but if no value is given then the default value is a blank string. Variables can be any of the standard data types, but the type is set automatically based on their value assignment. (Quoted values become strings, unquoted numeric values become numbers, etc.). Like parameters, variables are immutable. Once their value has been set, it cannot be changed.

Variables that are defined at the top-level of the script are known as global variables and can be referenced from within any of the script's templates or custom functions. The junos.xsl import file, which is imported when the standard script boilerplate is used, contains a default global variable that is available to all script types: **$junos-context**. Additional global variables can be created by defining them through the **var** statement at the top-level of the script.

```
var $connection = jcs:open();
var $user-name = "admin";
```

Template/local variables can be created by using the **var** statement within a template's or custom function's code block. Multiple **var** statements can be included, creating multiple variables, and they can occur anywhere within the template's code, but their usage is limited to the scope in which they are defined. For example, a variable that is defined at the template level can be used anywhere within the template's code, but a variable that is defined within a **for-each** code block within the template can only be used within that **for-each** loop.

```
template build-loop() {
    var $example1 = "This variable can be used throughout the template";
    ...
    for-each( ... ) {
        ...
        var $example2 = "This variable can only be used within the loop";
        ...
    }
    ...
}
```

Global variables can be overridden by template/local variables of the same name, but template/local variables cannot be overridden by each other, even if their scope is different:

Code example:

```
var $variable = "Global scope!";
match / {
    <op-script-results> {
        <output> "Before = " _ $variable;
        var $variable = "Local scope!";
        <output> "After = " _ $variable;
    }
}
```

Output:

```
Before = Global scope!
After = Local scope!
```

Example

This op script demonstrates the use of both global variables as well as template variables.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Global Variable - scope is the entire script */
var $connection = jcs:open();

match / {
    <op-script-results> {

        /* Variable scope is the <op-script-results> code block */
        var $users-rpc = {
            <get-system-users-information>;
        }

        /* $junos-context is a global variable defined in junos.xsl as of Junos 11.1 */
        if( $junos-context/tty ) {
            /* Variable scope is the if code block */
            var $result = jcs:execute( $connection, $users-rpc );
            var $user-entry = $result/uptime-information/user-table/
                user-entry[ tty == substring-after( $junos-context/tty, "/dev/tty" ) ];

            <output> "You logged in at: " _ $user-entry/login-time;
            <output> "From: " _ $user-entry/from;
        }
        else {
            <output> "TTY not available";
        }

        expr jcs:close( $connection );
    }
}
```

Output

```
You logged in at: 7:15AM
From: 10.0.0.50
```

version

Minimum Version: Junos 8.2

Syntax

version *1.0;*

Description

The **version** statement informs the script processor that SLAX version the script is coded in, and it must be the first statement of the script. Only comments are allowed to precede it.

All SLAX 1.0 scripts should report the version as 1.0.

Example

This simple op script demonstrates the use of a comment prior to the version statement.

Code

```
/* Comments can appear above version */
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> "Hello World!";
    }
}
```

Output

```
Hello World!
```

with

Minimum Version: Junos 8.2

Syntax

with *parameter-name;*
with *parameter-name* = *value;*

Description

The **with** statement is used with the **call** and **apply-templates** statements to pass parameter values to the invoked template. It is placed within the calling statement's code block and includes the parameter name and (optionally) the value that should be assigned:

```
call to-lower {
    with $string = "Junos";
}


apply-templates route {
    with $next-hop = "10.0.0.1";
}
```

When multiple parameters are assigned, their position relative to each other is insignificant because the template parameters are assigned based on their name rather than their position:

```
apply-templates route {
    with $next-hop = "10.0.0.1";
    with $metric = "10";
}
apply-templates route {
    with $metric = "10";
    with $next-hop = "10.0.0.1";
}
```

There is no need to specify a value for a parameter if the parameter is being assigned the value of a parameter or variable that shares the same name within the calling template; instead, when the parameter name is present in the **call** or **apply-templates** statement but lacks an assignment, SLAX automatically assumes that the parameter's value should be set to the value of the variable or parameter of the same name within the calling template. In other words, the following two template calls are identical:

```
var $string = "Junos";
call to-upper {
    with $string = $string;
}
call to-upper {
    with $string;
}
```

However, when assigning parameter values in this manner, it is still necessary to include the parameter name, without an assigned value, in the **call** or **apply-templates** statement; otherwise, if the parameter is not specified as part of the **call** or **apply-templates** statement then it is assigned its default value.

Example

This op script demonstrates how to pass parameter values to called template by using the **with** statement.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> {
            call append-string {
                with $string = "Junos";
                with $character = ".";
                with $length = 3;
            }
        }

        <output> {
            var $string = "Hello World";
            call append-string {
                with $string;
                with $length = 5;
                with $character = "!";
            }
        }
```

```
    }
}

/* Append variable number of  characters at end of string */
template append-string() {
    param $string;
    param $length = 1;
    param $character = "-";

    expr $string _ str:padding( $length, $character );
}
```

Output

```
Junos...
Hello World!!!!!
```

Part 2: Operators

The SLAX language contains a number of boolean, comparison, mathematic, and other operators. Most are derived from XPath and can be used in any context, but others were added as improvements within SLAX, and while these new operators improve the efficiency and attractiveness of writing SLAX scripts, there are scenarios where the SLAX-specific operators are not supported because the standard SLAX to XSLT conversion process is unable to correctly translate the SLAX operator into its XSLT equivalent.

These scenarios include:

- The XPath expression provided as an argument to the **dyn:evaluate**(), **saxon:evaluate**(), and **saxon:expression**() functions
- The attribute expression strings used by elements such as **<func:result>**, **<redirect:write>**, and **<xsl:sort>**.

Generally speaking, the SLAX-specific operators can never be used when the XPath expression is within a quoted string, rather than a part of the script code.

Addition

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number1 + *$number2*

Description

The plus sign "+" is the SLAX addition operator. It adds the two surrounding operands together and returns a number as a result. If either operand's data type is not a number, then it is converted into a number following the same process as the **number**() function. The addition operator comes from XPath, so there are no restrictions on its usage.

Example

This op script demonstrates the addition operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "Result is " _ $number1 + $number2;
    }
}
```

Output

```
jnpr@srx210> op addition
Number 1: 14
Number 2: 29
Result is 43
```

And (SLAX)

Source: SLAX
Minimum Version: Junos 8.2

Syntax

expression && *expression*

Description

SLAX introduced the double ampersand 'and' operator "&&" in order to match the syntax of other common programming languages; however, the standard XPath 'and' operator, which is the lowercase word "and", can be used within SLAX scripts as well.

This operator gives a boolean result of true or false, with true indicating that both surrounding expressions are true and false indicating otherwise. The expressions could be single objects, the results of other operators, or more complex expressions, but in all cases the computation is based on their boolean value, so a conversion to boolean, as if by calling the **boolean**() function, is performed if necessary.

Code example:

```
<output> ( 5 < 10 ) && "Not empty string";
```

Output:

```
true
```

If the left expression evaluates to false then it is impossible for the 'and' operator to evaluate to true, so the right expression is not even evaluated.

This 'and' operator comes from SLAX, so it cannot be used in circumstances where only XSLT/XPath syntax is allowed, such as within expression strings. In those scenarios, the XPath 'and' operator should be used instead.

Example

This op script demonstrates the SLAX 'and' operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $day = date:day-name();
        var $hour = date:hour-in-day();

        <output> "Today is " _ $day;
        <output> "The hour is " _ $hour;
```

```
        if( $day == "Saturday" && $hour < 12 ) {
            <output> "It is Saturday morning, why are you working?!";
        }
    }
}
```

Output

```
Today is Saturday
The hour is 9
It is Saturday morning, why are you working?!
```

And (XPath)

Source: XPath
Minimum Version: Junos 8.2

Syntax

expression **and** *expression*

Description

The XPath 'and' operator is the lowercase word "and". A SLAX-specific variation of the 'and' operator is available as well, the double ampersand: "&&".

This operator gives a boolean result of true or false, with true indicating that both surrounding expressions are true and false indicating otherwise. The expressions could be single objects, the results of other operators, or more complex expressions, but in all cases the computation is based on their boolean value, so a conversion to boolean, as if by calling the **boolean**() function, is performed if necessary.

Code example:

```
<output> ( 5 < 10 ) and "Not empty string";
```

Output:

```
true
```

If the left expression evaluates to false then it is impossible for the 'and' operator to evaluate to true, so the right expression is not even evaluated. This 'and' operator comes from XPath, so there are no restrictions on its use.

Example

This op script demonstrates the XPath 'and' operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
```

```
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $month = date:month-name();
        var $day = date:day-in-month();

        <output> "The day is " _ $day;
        <output> "The month is " _ $month;

        if( $day == 1 and $month == "April" ) {
            <output> "April Fools!";
        }
    }
}
```

Output

```
The day is 1
The month is April
April Fools!
```

Assignment

Source: SLAX
Minimum Version: Junos 8.2

Syntax

var $variable = value;

Description

The equals sign "=" is the SLAX assignment operator. It is used to assign a variable's value or a parameter's default value when they are defined, to assign a namespace to a prefix, and to indicate what value template parameters should be assigned when a template is called.

```
/* The variable $example is assigned the string value of "admin" */
var $example = "admin";


/* The print-message template is called and its message parameter
   is set to "Hello World" */
call print-message( $message = "Hello World" );
```

The assignment operator was introduced by SLAX, so it would be unavailable within expression strings that only allow XSLT/XPath sytax; however, variable initialization and template calls are not allowed within those types of expressions any way.

Example

This op script demonstrates the assignment operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $value = "I was assigned this value";

        <output> $value;
    }
}
```

Output

```
I was assigned this value
```

Division

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number1 **div** *$number2*

Description

The word “div” is the SLAX division operator. It divides the left operand by the right operand, using floating-point division, and returns a number as a result. If either operand’s data type is not a number, then it is converted into a number following the same process as the **number**() function. The division operator comes from XPath, so there are no restrictions on its usage.

Many other programming languages use the forward-slash “/” for division; however, this is used for location paths within SLAX and therefore is not available for use as the division operator.

Example

This op script demonstrates the division operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number1: ");
        var $number2 = jcs:get-input( "Number2: ");

        <output> "Result: " _ $number1 div $number2;
    }
}
```

Output

```
jnpr@srx210> op division
Number1: 7
Number2: 2
Result: 3.5
```

Equality (SLAX)

Source: SLAX
Minimum Version: Junos 8.2

Syntax

$object == *$object*

Description

SLAX introduced the double equal sign equality operator "==" in order to match the syntax of other common programming languages and thereby allow script writers to maintain the habit of using "==" as an equality test; however, the standard XPath equality operator, which is a single equal sign "=", can be used within SLAX scripts as well. The equality test results in a boolean result of true or false, based on whether the two objects are equal or not. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To test for equality, the first matching scenario is used:

- Both objects are node-sets
  - The objects are considered equal if the string-value of at least one node in the first node-set is equal to the string-value of at least one node in the other node-set.
- One object is a node-set and the other is a number
  - The objects are considered equal if the number is equal to at least one node within the node-set, after the string-value of that node has been converted to a number through the **number**() function.
- One object is a node-set and the other is a string

- The objects are considered equal if the string-value of at least one node in the node-set is equal to the string's value.

- One object is a node-set and the other is a boolean
  - The node-set is converted to its boolean value through the **boolean**() function. The objects are considered equal if this boolean value equals the value of the boolean object.
- At least one object is a boolean
  - The objects are considered equal if the boolean values are equal after both objects have been converted to boolean via the **boolean**() function (if they are not already booleans).
- At least one object is a number
  - The objects are considered equal if the numbers are equal after both objects have been converted to a number via the **number**() function (if they are not already numbers).
- At least one object is a string
  - Otherwise, the objects are converted to strings via the **string**() function (if they are not already strings) and are considered to be equal if their string-values are equal.

String comparisons are always case sensitive, and null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing.

This equality operator comes from SLAX, so it cannot be used in circumstances where only XSLT/XPath syntax is allowed such as within expression strings. In those scenarios, the XPath equality operator should be used instead.

Example

This op script demonstrates the SLAX equality operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $string = jcs:get-input( "String to compare against boolean false: " );
        expr jcs:output( "Result: ", $string == false() );
        var $string2 = jcs:get-input( "String to compare against the number 12345: " );
        expr jcs:output( "Result: ", $string2 == 12345 );
        var $string3 = jcs:get-input(
          "String to compare against node-set with nodes: <node> 'abc' and <node> '123': " );
        var $ns-raw := {
            <node> "abc";
            <node> "123";
        }
        var $ns = $ns-raw/node;
        expr jcs:output( "Result: ", $string3 == $ns );
```

```
    }
}
```

Output

```
jnpr@srx210> op equality_slax
String to compare against boolean false: false
Result: false
String to compare against the number 12345: 12345
Result: true
String to compare against node-set with nodes: <node> 'abc' and <node> '123': abc
Result: true
```

Equality (XPath)

Source: XPath
Minimum Version: Junos 8.2

Syntax

$object = *$object*

Description

A single equal sign "=" is the XPath equality operator, which can be used within SLAX scripts, but the SLAX equality operator "==" is preferred in most cases. The equality test results in a boolean result of true or false, based on if the two objects are equal or not. The comparison performed depends on the data types of the two objects, but they both do not have to be the same data type in order to be compared. To test for equality, the first matching scenario is used:

- Both objects are node-sets
  - The objects are considered equal if the string-value of at least one node in the first node-set is equal to the string-value of at least one node in the other node-set
- One object is a node-set and the other is a number
  - The objects are considered equal if the number is equal to at least one node within the node-set, after the string-value of that node has been converted to a number through the **number**() function.
- One object is a node-set and the other is a string
  - The objects are considered equal if the string-value of at least one node in the node-set is equal to the string's value.
- One object is a node-set and the other is a boolean
  - The node-set is converted to its boolean value through the **boolean**() function. The objects are considered equal if this boolean value equals the value of the boolean object.
- At least one object is a boolean
  - The objects are considered equal if the boolean values are equal after both objects have been converted to boolean via the **boolean**() function (if they are not already booleans).

- At least one object is a number
  - The objects are considered equal if the numbers are equal after both objects have been converted to a number via the **number**() function (if they are not already numbers).
- At least one object is a string
  - Otherwise, the objects are converted to strings via the **string**() function (if they are not already strings) and are considered to be equal if their string-values are equal.

String comparisons are always case sensitive, and null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing.

This equality operator comes from XPath, so it has no restrictions on its use. In contrast, the SLAX equality operator cannot be used within expression strings because they are not converted as part of the automatic SLAX to XSLT conversion process.

Example

This op script demonstrates a scenario where the XPath equality operator must be used rather than the SLAX equality operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns dyn = "http://exslt.org/dynamic";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $name1 = "admin";
        var $name2 = "jnpr";
        var $name3 = "admin";
        var $name4 = "operator";

        expr jcs:output( "Select which variables to compare: " );
        expr jcs:output( "$name1 = \"", $name1, "\"" );
        expr jcs:output( "$name2 = \"", $name2, "\"" );
        expr jcs:output( "$name3 = \"", $name3, "\"" );
        expr jcs:output( "$name4 = \"", $name4, "\"" );

        var $var1 = jcs:get-input( " \nFirst variable: " );
        var $var2 = jcs:get-input( "Second variable: " );

        /* Process the expression dynamically, but the SLAX == operator cannot be
           used within this expression string, so the XPath = operator must be used
           instead */
        var $result = dyn:evaluate( concat( $var1, " = ", $var2 ) );
        expr jcs:output( "_\r \n", "Result: ", $result );
    }
}
```

Output

```
jnpr@srx210> op equality_xpath
Select which variables to compare:
$name1 = "admin"
$name2 = "jnpr"
$name3 = "admin"
$name4 = "operator"
First variable: $name1
Second variable: $name2

Result: false

jnpr@srx210> op equality_xpath
Select which variables to compare:
$name1 = "admin"
$name2 = "jnpr"
$name3 = "admin"
$name4 = "operator"
First variable: $name1
Second variable: $name3

Result: true
```

Greater than

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number > *$number*

Description

The greater than sign “>” is the SLAX greater than operator. It gives a boolean result of true or false; with true indicating that the left object is greater than the right object and false indicating otherwise. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To perform the comparison, the first matching scenario is used:

- Both objects are node-sets
  - The left node-set is considered to be greater than the right node-set if there is at least one node, converted into a number, within the left node-set that is greater than at least one node, converted into a number, within the right node-set.
- One object is a node-set and the other is a number
  - The node-set is the left object
    - The node-set is considered to be greater than the number object if any node, converted into a number, within the node-set is greater than the number object.
  - The number is the left object

        - The number object is considered to be greater than the node-set if the number object is greater than any node, converted into a number, within the node-set.
- One object is a node-set and the other is a string
    - The node-set is the left object
        - The node-set is considered to be greater than the string object if any node, converted into a number, is greater than the string object, converted into a number.
    - The string is the left object
        - The string object is considered to be greater than the node-set if the string object, converted into a number, is greater than any node, converted into a number, within the node-set.
- One object is a node-set and the other is a boolean
    - The node-set is converted into a boolean, which is then converted into a number. The boolean object is also converted into a number. The left number is then compared to the right number to determine if it is greater than the other.
- Otherwise, both objects are converted into numbers (if they are not already numbers) and the left number is compared to the right number to determine if it is greater than the other.

Due to the logic followed with node-set comparisons, if at least one of the objects being compared is a node-set then it is possible that both "$object1 > $object2" and "$object2 > $object1" will evaluate to true; however, null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing. The greater than operator comes from XPath, so it has no restrictions on its use.

Example

This op script demonstrates how to use the greater than operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "( " _ $number1 _ " > " _ $number2 _ " ) = " _ $number1 > $number2;
    }
}
```

Output

```
jnpr@srx210> op greater_than
Number 1: 15
```

```
Number 2: 16
( 15 > 16 ) = false
```

Greater than or equal to

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number >= *$number*

Description

The character combination ">=" is the SLAX greater than or equal to operator. It gives a boolean result of true or false, with true indicating that the left object is greater than or equal to the right object and false indicating otherwise. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To perform the comparison, the first matching scenario is used:

- Both objects are node-sets
  - The left node-set is considered to be greater than or equal to the right node-set if there is at least one node, converted into a number, within the left node-set that is greater than or equal to at least one node, converted into a number, within the right node-set.
- One object is a node-set and the other is a number
  - The node-set is the left object
    - The node-set is considered to be greater than or equal to the number object if any node, converted into a number, within the node-set is greater than or equal to the number object.
  - The number is the left object
    - The number object is considered to be greater than or equal to the node-set if the number object is greater than or equal to any node, converted into a number, within the node-set.
- One object is a node-set and the other is a string
  - The node-set is the left object
    - The node-set is considered to be greater than or equal to the string object if any node, converted into a number, is greater than or equal to the string object, converted into a number.
  - The string is the left object
    - The string object is considered to be greater than or equal to the node-set if the string object, converted into a number, is greater than or equal to any node, converted into a number, within the node-set.
- One object is a node-set and the other is a boolean

  - The node-set is converted into a boolean, which is then converted into a number. The boolean object is also converted into a number. The left number is then compared to the right number to determine if it is greater than or equal to the other.
- Otherwise, both objects are converted into numbers (if they are not already numbers) and the left number is compared to the right number to determine if it is greater than or equal to the other.

Null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing. The greater than or equal to operator comes from XPath, so it has no restrictions on its use.

Example

This op script demonstrates how to use the greater than or equal to operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "( " _ $number1 _ " >= " _ $number2 _ " ) = " _ $number1 >= $number2;
    }
}
```

Output

```
jnpr@srx210> op greater_than_or_equal_to
Number 1: 1500.2314
Number 2: 1500.2314
( 1500.2314 >= 1500.2314 ) = true
```

Inequality

Source: XPath
Minimum Version: Junos 8.2

Syntax

$object **!=** *$object*

Description

The character combination "!=" is the SLAX inequality operator. It gives a boolean result of true or false; with true indicating that the objects are not equal and false indicating otherwise. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To test for inequality, the first matching scenario is used:

- Both objects are node-sets
  - The objects are considered to not be equal if the string-value of at least one node in the first node-set is not equal to the string-value of at least one node in the other node-set.
- One object is a node-set and the other is a number
  - The objects are considered to not be equal if the number is not equal to at least one node within the node-set, after the string-value of that node has been converted to a number through the **number**() function.
- One object is a node-set and the other is a string
  - The objects are considered to not be equal if the string-value of at least one node in the node-set is not equal to the string's value.
- One object is a node-set and the other is a boolean
  - The node-set is converted to its boolean value through the **boolean**() function. The objects are considered to not be equal if this boolean value does not equal the value of the boolean object.
- At least one object is a boolean
  - The objects are considered to not be equal if the boolean values are not equal after both objects have been converted to boolean via the **boolean**() function (if they are not already booleans).
- At least one object is a number
  - The objects are considered to not be equal if the numbers are not equal after both objects have been converted to a number via the **number**() function (if they are not already numbers).
- At least one object is a string
  - Otherwise, the objects are converted to strings via the **string**() function (if they are not already strings) and are considered to not be equal if their string-values are not equal.

String comparisons are always case sensitive, and null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing. The inequality operator comes from XPath, so it has no restrictions on its use.

Example

This op script demonstrates how to use the inequality operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $string = jcs:get-input( "String to compare against boolean false: " );
        expr jcs:output( "Result: ", $string != false() );
        var $string2 = jcs:get-input( "String to compare against the number 12345: " );
        expr jcs:output( "Result: ", $string2 != 12345 );
        var $string3 = jcs:get-input(
          "String to compare against node-set with nodes: <node> 'abc' and <node> '123': " );
        var $ns-raw := {
            <node> "abc";
            <node> "123";
        }
        var $ns = $ns-raw/node;
        expr jcs:output( "Result: ", $string3 != $ns );
    }
}
```

Output

```
jnpr@srx210> op inequality
String to compare against boolean false: false
Result: true
String to compare against the number 12345: 12345
Result: false
String to compare against node-set with nodes: <node> 'abc' and <node> '123': no-match
Result: true
```

Less than

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number < *$number*

Description

The less than sign "<" is the SLAX less than operator. It gives a boolean result of true or false; with true indicating that the left object is less than the right object and false indicating otherwise. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To perform the comparison, the first matching scenario is used:

- Both objects are node-sets
  - The left node-set is considered to be less than the right node-set if there is at least one node, converted into a number, within the left node-set that is less than at least one node, converted into a number, within the right node-set.
- One object is a node-set and the other is a number
  - The node-set is the left object

        - The node-set is considered to be less than the number object if any node, converted into a number, within the node-set is less than the number object.
    - The number is the left object
        - The number object is considered to be less than the node-set if the number object is less than any node, converted into a number, within the node-set.
- One object is a node-set and the other is a string
    - The node-set is the left object
        - The node-set is considered to be less than the string object if any node, converted into a number, is less than the string object, converted into a number.
    - The string is the left object
        - The string object is considered to be less than the node-set if the string object, converted into a number, is less than any node, converted into a number, within the node-set.
- One object is a node-set and the other is a boolean
    - The node-set is converted into a boolean, which is then converted into a number. The boolean object is also converted into a number. The left number is then compared to the right number to determine if it is less than the other.
- Otherwise, both objects are converted into numbers (if they are not already numbers) and the left number is compared to the right number to determine if it is less than the other.

Due to the logic followed with node-set comparisons, if at least one of the objects being compared is a node-set then it is possible that both "$object1 < $object2" and "$object2 < $object1" will evaluate to true; however, null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing. The less than operator comes from XPath, so it has no restrictions on its use.

Example

This op script demonstrates how to use the less than operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "( " _ $number1 _ " < " _ $number2 _ " ) = " _ $number1 < $number2;
    }
}
```

Output

```
jnpr@srx210> op less_than
Number 1: 5123
Number 2: 6000
( 5123 < 6000 ) = true
```

Less than or equal to

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number **<=** *$number*

Description

The character combination "<=" is the SLAX less than or equal to operator. It gives a boolean result of true or false; with true indicating that the left object is less than or equal to the right object and false indicating otherwise. The comparison performed depends on the data types of the two objects, but they do not both have to be the same data type in order to be compared. To perform the comparison, the first matching scenario is used:

- Both objects are node-sets
  - The left node-set is considered to be less than or equal to the right node-set if there is at least one node, converted into a number, within the left node-set that is less than or equal to at least one node, converted into a number, within the right node-set.
- One object is a node-set and the other is a number
  - The node-set is the left object
    - The node-set is considered to be less than or equal to the number object if any node, converted into a number, within the node-set is less than or equal to the number object.
  - The number is the left object
    - The number object is considered to be less than or equal to the node-set if the number object is less than or equal to any node, converted into a number, within the node-set.
- One object is a node-set and the other is a string
  - The node-set is the left object
    - The node-set is considered to be less than or equal to the string object if any node, converted into a number, is less than or equal to the string object, converted into a number.
  - The string is the left object
    - The string object is considered to be less than or equal to the node-set if the string object, converted into a number, is less than or equal to any node, converted into a number, within the node-set.

- One object is a node-set and the other is a boolean
  - The node-set is converted into a boolean, which is then converted into a number. The boolean object is also converted into a number. The left number is then compared to the right number to determine if it is less than or equal to the other.
- Otherwise, both objects are converted into numbers (if they are not already numbers) and the left number is compared to the right number to determine if it is less than or equal to the other.

Null node-sets always evaluate to false unless they are being compared to a boolean object, because they have no nodes to compare the other object against. Objects with an external data type (stored expressions) cannot be compared at the time of this writing. The less than or equal to operator comes from XPath, so it has no restrictions on its use.

Example

This op script demonstrates how to use the less than or equal to operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "( " _ $number1 _ " <= " _ $number2 _ " ) = " _ $number1 <= $number2;
    }
}
```

Output

```
jnpr@srx210> op less_than_or_equal_to
Number 1: 5319213
Number 2: 5319213
( 5319213 <= 5319213 ) = true
```

Location path step

Source: XPath
Minimum Version: Junos 8.2

Syntax

location-path-step/location-path-step

Description

The forward slash "/" is the SLAX location path step operator, which indicates a new step in a location path. Location paths are used to extract data from a XML document, with each step parsing further into the document along one of multiple axes.

Code example:

```
var $data := {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
            <unit> {
                <name> "0";
            }
        }
    }
}
<output> $data/interfaces/interface/unit/name;
```

Output:

```
0
```

In the example above, the location path is based on the $data node-set, which consists of a single root node for the embedded XML data structure. The "/" operator is used to indicate a step, using the default child axis, which selects all interfaces nodes (in this example there is only one). The "/" operator is then used again to take another step, using the default child axis once again, but this time selecting any <interface> nodes. The "/" operator is used two more times, each time utilizing the child axis, moving further into the XML document. The next step selects the <unit> node, and then the final step selects the <name> node. Once completed, the location path result is a node-set that consists of a single node: the desired logical interface's <name> node.

If the "/" operator appears at the beginning of the location path then the location path is an absolute path, based on the root node of the current node; otherwise, it is a relative path, which could be based on the context node or on a node-set variable. The "/" operator comes from XPath, so there are no restrictions on its use.

Example

This op script demonstrates the use of the "/" operator within a location path that utilizes both the child and attribute axes.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve system uptime */
        var $uptime = jcs:invoke( "get-system-uptime-information" );

        /* Extract the seconds */
```

```
            var $seconds = $uptime/current-time/date-time/@junos:seconds;

            <output> "Current seconds: " _ $seconds;

            /* Record XML results for comparison */
            <xsl:document href="/var/tmp/uptime-output" indent="yes"> {
                copy-of $uptime;
            }
        }
    }
```

Output

```
    jnpr@srx210> op location_path_step
    Current seconds: 1304339621

    jnpr@srx210> file show /var/tmp/uptime-output
    <?xml version="1.0"?>
    <system-uptime-information xmlns="http://xml.juniper.net/junos/11.1R1/junos"
xmlns:junos="http://xml.juniper.net/junos/*/junos">
    <current-time>
    <date-time junos:seconds="1304339621">2011-05-02 12:33:41 UTC</date-time>
    </current-time>
    <system-booted-time>
    <date-time junos:seconds="1304313164">2011-05-02 05:12:44 UTC</date-time>
    <time-length junos:seconds="26457">07:20:57</time-length>
    </system-booted-time>
    <protocols-started-time>
    <date-time junos:seconds="1304313352">2011-05-02 05:15:52 UTC</date-time>
    <time-length junos:seconds="26269">07:17:49</time-length>
    </protocols-started-time>
    <last-configured-time>
    <date-time junos:seconds="1304339357">2011-05-02 12:29:17 UTC</date-time>
    <time-length junos:seconds="264">00:04:24</time-length>
    <user>jnpr</user>
    </last-configured-time>
    <uptime-information>
    <date-time junos:seconds="1304339621">12:33PM</date-time>
    <up-time junos:seconds="26487">7:21</up-time>
    <active-user-count junos:format="2 users">2</active-user-count>
    <load-average-1>0.08</load-average-1>
    <load-average-5>0.06</load-average-5>
    <load-average-15>0.02</load-average-15>
    </uptime-information>
    </system-uptime-information>
```

Location path multiple step

Source: XPath
Minimum Version: Junos 8.2

Syntax

location-path-step//location-path-step

Description

The double forward slash "//" is the SLAX location path multiple step operator, which indicates that all descendants of the context node, along with the context node itself, should be considered as the context of the following location path step. The "//" operator is an abbreviation of "/descendant-or-self::node()/.

Code example:

```
var $data := {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
            <unit> {
                <name> "0";
            }
        }
    }
}
for-each( $data//name ) {
    <output> .;
}
```

Output:

```
ge-0/0/0
0
```

In the example above, the location path is based on the $data node-set, which consists of a single root node for the embedded XML data structure. The "//" operator is used to indicate that all descendants of this root node, and the root node itself, should be considered as context for the following step, which checks the child axis for a <name> name. The result is a node-set that consists of two <name> nodes: one that is the child of <interface> and the other than is the child of <unit>.

If the "//" operator appears at the beginning of the location path then the location path is an absolute path, based on the root node of the current node; otherwise, it is a relative path, which could be based on the context node or on a node-set variable. The "//" operator comes from XPath, so there are no restrictions on its use.

Example

This op script demonstrates the use of the "//" operator within a location path to evaluate the following location path step over the context node as well as all of its descendants.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $data := {
            <interfaces> {
                <interface> {
                    <name> "ge-0/0/0";
```

```
                    <unit> {
                        <name> "0";
                        <family> {
                            <inet> {
                                <address> {
                                    <name> "10.0.0.1/24";
                                }
                            }
                        }
                    }
                }
                <interface> {
                    <name> "ge-1/0/0";
                    <unit> {
                        <name> "0";
                        <family> {
                            <inet> {
                                <address> {
                                    <name> "10.1.0.25/24";
                                }
                            }
                        }
                    }
                }
            }
        }

        /* Display the two interface addresses */
        for-each( $data//address/name ) {
            <output> .;
        }
    }
}
```

Output

```
10.0.0.1/24
10.1.0.25/24
```

Modulo

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number1 **mod** *$number2*

Description

The word “mod” is the SLAX modulo operator. It returns the remainder after dividing the left operand by the right operand, using truncating division. The result is a number and its sign matches that of the dividends (the left operand). If either operand’s data type is not a number, then it is converted into a number following the same process as the **number**() function. The modulo operator comes from XPath, so there are no restrictions on its usage.

Example

This op script demonstrates the modulo operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number1: ");
        var $number2 = jcs:get-input( "Number2: ");

        <output> "Result: " _ $number1 mod $number2;
    }
}
```

Output

```
jnpr@srx210> op modulo
Number1: 99
Number2: 6
Result: 3
```

Multiplication

Source: XPath
Minimum Version: Junos 8.2

Syntax

*$number1 * $number2*

Description

The asterisk "*" is the SLAX multiplication operator. It multiplies the two surrounding operands together and returns a number as a result. If either operand's data type is not a number then it is converted into a number following the same process as the **number**() function. The multiplication operator comes from XPath, so there are no restrictions on its usage.

Example

This op script demonstrates the multiplication operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number1: ");
        var $number2 = jcs:get-input( "Number2: ");

        <output> "Result: " _ $number1 * $number2;
    }
}
```

Output

```
jnpr@srx210> op multiplication
Number1: 3
Number2: 9
Result: 27
```

node-set conversion

Source: SLAX
Minimum Version: Junos 9.2

Syntax

var $node-set := *{*
node contents...
}

Description

The character combination ":=" is the SLAX node-set conversion operator. When it is used in place of the assignment operator "=", it causes a variable to be initialized as a node-set rather than a result tree fragment, which allows the variable's contents to be accessible through location paths.

When using the assignment operator, anytime curly brackets are used to enclose a variable assignment or an element is assigned directly to a variable, the variable's data type is a result tree fragment:

Code example:

```
var $rtf1 = <command> "clear bgp neighbor";
var $configuration = {
    <configuration> {
        <system> {
            <host-name> "new-name";
        }
    }
```

```
}
var $connection = jcs:open();
var $rtf2 = { call jcs:load-configuration( $connection, $configuration); }
<output> "$rtf1 is a " _ exsl:object-type( $rtf1 );
<output> "$configuration is a " _ exsl:object-type( $configuration );
<output> "$rtf2 is a " _ exsl:object-type( $rtf2 );
```

Output:

```
$rtf1 is a RTF
$configuration is a RTF
$rtf2 is a RTF
```

The problem with result tree fragments is that their contents cannot be parsed by location paths, which means that in the example above the $rtf2 variable, assigned to the results of the **jcs:load-configuration** template, cannot be checked for <xnm:error> nodes. This is a common problem when working with template results, because when template results are stored in a variable the variable's data type will always be result tree fragment if the normal assignment operator is used. A better solution would be to assign the results to a node-set variable by using the node-set conversion operator instead of the assignment operator. That way the results variable is accessible to location paths:

```
var $ns := { call jcs:load-configuration( $connection, $configuration); }
```

Now, the $ns variable is a node-set, and can be parsed as desired:

```
If( $ns//self::xnm:error ) {
    ...
}
```

The node-set conversion operator is not needed for function results because, unlike templates, they return their XML results as node-sets, so no conversion is necessary:

Code example:

```
/* The assignment operator should be used with functions */
var $tokens = str:tokenize( $hostname, "" );
<output> "$tokens is a " _ exsl:object-type( $tokens );
<output> "$hostname = " _ $hostname;
for-each( $tokens ) {
    <output> .;
}
```

Output:

```
$tokens is a node-set
$hostname = srx210
s
r
x
2
1
0
```

Typically, the node-set conversion operator is used to assign the XML data directly:

Code example:

```
var $ns1 := <command> "show version";
var $ns2 := {
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/0";
}
<output> "$ns1 is a " _ exsl:object-type( $ns1 );
<output> "$ns2 is a " _ exsl:object-type( $ns2 );
```

Output:

```
$ns1 is a node-set
$ns2 is a node-set
```

But the node-set conversion operator can also be used to convert an existing result tree fragment variable's contents into a node-set by using the **copy-of** statement within curly brackets:

Code example:

```
var $configuration = {
    <configuration> {
        <system> {
            <host-name> "new-name";
        }
    }
}
var $connection = jcs:open();
var $rtf = { call jcs:load-configuration( $connection, $configuration); }
var $results := { copy-of $rtf; }
<output> "$configuration is a " _ exsl:object-type( $configuration );
<output> "$rtf is a " _ exsl:object-type( $rtf );
<output> "$results is a " _ exsl:object-type( $results );
```

Output:

```
$configuration is a RTF
$rtf is a RTF
$results is a node-set
```

The node-set conversion operator works by creating a node-set that consists of a single node, which is the root node of the XML contents. This is a different reference point than what is typically seen in node-sets returned by functions, which usually contain one or more element nodes rather than a single root node, so location paths need to be designed to account for the different starting location:

Code example:

```
var $data := {
    <interface> "ge-0/0/0";
    <interface> "fe-2/0/0";
    <interface> "so-2/1/0";
}
<output> "$data is a " _ exsl:object-type( $data );
<output> "$data contains " _ count( $data ) _ " nodes";
<output> "$data/* contains " _ count( $data/* ) _ " nodes";
for-each( $data/interface ) {
    <output> .;
}
```

Output:

```
$data is a node-set
$data contains 1 nodes
$data/* contains 3 nodes
ge-0/0/0
fe-2/0/0
so-2/1/0
```

In the above example, a script writer might believe erroneously that the $data node-set would consist of three <interface> nodes, but as the example shows, the $data node-set consists of a single node, which is the root node, and the root node has three <interface> child nodes, which can be accessed through a location path of $data/interface.

Because it is a SLAX-specific operator, the node-set conversion operator cannot be used in circumstances where only XSLT/XPath syntax is permitted, such as in expression strings; however, variable assignment is not allowed in those circumstances any way.

Example

This op script demonstrates the node-set conversion operator. The script uses this operator to store the **jcs:load-configuration** template results in a node-set variable rather than a result tree fragment.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve new hostname */
        var $new-name = jcs:get-input( "Enter new hostname: " );

        /* Build config change - This is a RTF, but that is fine because it
           does not need to be parsed by location paths. */
        var $configuration = {
            <configuration> {
                <system> {
                    <host-name> $new-name;
                }
            }
        }

        /* Open the management connection */
        var $connection = jcs:open();

        /* Make the change and assign the results to a node-set variable */
        var $results := { call jcs:load-configuration( $connection, $configuration ); }

        /* Check for errors or report success */
        if( $results//self::xnm:error ) {
            /* Copy any errors to the result tree */
            copy-of $results;
        }
        else {
            <output> "Success!";
        }
    }
}
```

Output

```
jnpr@new-name> op node-set_conversion
Enter new hostname: srx210
Success!

jnpr@new-name>

jnpr@srx210>
```

Or (SLAX)

Source: SLAX
Minimum Version: Junos 8.2

Syntax

expression || *expression*

Description

SLAX introduced the double vertical bar 'or' operator "||" in order to match the syntax of other common programming languages; however, the standard XPath 'or' operator, which is the lowercase word "or", can be used within SLAX scripts as well.

This operator gives a boolean result of true or false, with true indicating that at least one of the two surrounding expressions are true and false indicating otherwise. The expressions could be single objects, the results of other operators, or more complex expressions, but in all cases the computation is based on their boolean value, so a conversion to boolean, as if by calling the **boolean**() function, is performed if necessary.

Code example:

```
<output> (( 1 + 1 ) == 3) || string-length("abcd") == 4;
```

Output:

```
true
```

If the left expression evaluates to true then it is impossible for the 'or' operator to evaluate to false, so the right expression is not even evaluated.

This 'or' operator comes from SLAX, so it cannot be used in circumstances where only XSLT/XPath syntax is allowed such as within expression strings. In those scenarios, the XPath 'or' operator should be used instead.

Example

This op script demonstrates the SLAX 'or' operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $day = date:day-name();

        <output> "The day is " _ $day;

        if( $day == "Saturday" || $day == "Sunday" ) {
```

```
            <output> "It's the weekend!";
        }
    }
}
```

Output

```
The day is Saturday
It's the weekend!
```

Or (XPath)

Source: XPath
Minimum Version: Junos 8.2

Syntax

expression **or** *expression*

Description

The XPath 'or' operator is the lowercase word "or". Alternatively, the SLAX-specific 'or' operator can be used as well, which is the double vertical bar "||".

This operator gives a boolean result of true or false, with true indicating that at least one of the two surrounding expressions are true and false indicating otherwise. The expressions could be single objects, the results of other operators, or more complex expressions, but in all cases the computation is based on their boolean value, so a conversion to boolean, as if by calling the **boolean**() function, is performed if necessary.

Code example:

```
<output> (( 1 + 1 ) == 3) or string-length("abcd") == 4;
```

Output:

```
true
```

If the left expression evaluates to true then it is impossible for the 'or' operator to evaluate to false, so the right expression is not even evaluated. This 'or' operator comes from XPath , so there are no restrictions on its use.

Example

This op script demonstrates the XPath 'or' operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";
```

```
match / {
    <op-script-results> {

        var $day = date:day-name();

        <output> "The day is " _ $day;

        if( $day == "Saturday" or $day == "Sunday" ) {
            <output> "It's the weekend!";
        }
    }
}
```

Output

```
The day is Sunday
It's the weekend!
```

String concatenation

Source: SLAX
Minimum Version: Junos 8.2

Syntax

$string1 _ $string2

Description

The underscore: "_" is the SLAX string concatenation operator. Although some other programming languages can concatenate strings with their addition operator, in SLAX the addition operator always converts its operands into numbers, so a separate string concatenation operator is necessary to combine the two strings into one combined string. If the operands are not currently strings then they are converted into a string before being concatenated together.

Code example:

```
<output> "$user = " _ $user;
```

Output:

```
$user = jnpr
```

Because it is a SLAX-specific operator, the string concatenation operator cannot be used in circumstances where only XSLT/XPath syntax is permitted, such as in expression strings, and there is no XPath equivalent for this operator, so when SLAX-specific operators are not permitted the **concat**() function would have to be used instead. Also, at the time of this writing, the string concatenation operator cannot be used when assigning arguments to a function. For example, it is not possible to do: jcs:get-input($string1 _ $string2); instead, the **concat**() function would be necessary: jcs:get-input(concat($string1, $string2)).

Example

This op script demonstrates the concatenation operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns dyn = "http://exslt.org/dynamic";

import "../import/junos.xsl";

match / {
    <op-script-results> {

    var $parameters = "Parameters:\n" _
                      "$user, $script, $hostname, $product,\n" _
                      "$localtime, $localtime-iso";
    expr jcs:output( $parameters );
    var $result = jcs:get-input( "Display which parameter?: " );

    /* The SLAX string concatenation operator cannot be used in the XPath string
       provided to dyn:evaluate(), so the concat() function is used instead */
    <output> dyn:evaluate(
                concat( "concat( 'Value of ', $result, ' is ',", $result, ")"));

    }
}
```

Output

```
jnpr@srx210> op string_concatenation
Parameters:
$user, $script, $hostname, $product,
$localtime, $localtime-iso
Display which parameter?: $script
Value of $script is string_concatenation.slax
```

Subtraction

Source: XPath
Minimum Version: Junos 8.2

Syntax

$number1 - $number2

Description

The minus sign:"-" between two operands, is the SLAX subtraction operator. It subtracts the right operand from the left operand and returns a number as a result. If either operand's data type is not a number, then it is converted into a number following the same process as the **number**() function. Because the "-" character is allowed within names it can only be treated as the subtraction operator if it is preceded by whitespace. The subtraction operator comes from XPath, so there are no restrictions on its usage.

Example

This op script demonstrates the subtraction operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number1 = jcs:get-input( "Number 1: ");
        var $number2 = jcs:get-input( "Number 2: ");

        <output> "Result is " _ $number1 - $number2;
    }
}
```

Output

```
jnpr@srx210> op subtraction
Number 1: 97
Number 2: 11
Result is 86
```

Unary Minus

Source: XPath
Minimum Version: Junos 8.2

Syntax

- *$number*

Description

The minus sign "-" with only a following operand, is the SLAX unary minus. It changes the sign of the number from positive to negative or from negative to positive. If the operand is not a number, then it is converted into a number following the same process as the **number**() function. Because the "-" character is allowed within names it can only be treated as the unary minus operator if it is preceded by whitespace. The unary minus operator comes from XPath, so there are no restrictions on its usage.

While it might seem strange to consider -5 an example of an operation rather than just a negative number, consider more complex examples such as -(15 - 9) or --102, which give answers of -6 and (positive) 102 respectively.

Example

This op script demonstrates the unary minus operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $number = jcs:get-input( "Number: ");

        <output> "-(" _$number _ ") is " _ - $number;
    }
}
```

Output

```
jnpr@srx210> op unary_minus
Number: -19
-(-19) is 19
```

Union

Source: XPath
Minimum Version: Junos 8.2

Syntax

$node-set | *$node-set*

Description

The single vertical bar "|" is the SLAX union operator. It returns the union of the two node-sets, which is the combined set of unique nodes from the two node-sets.

Because node-sets cannot contain duplicate nodes, the union operator will never cause two references to the same node to be present within the result node-set, even if both node-set operands contain the identical node; however, this does not prevent nodes with the same name and value from being present in the node-set, so long as they come from different positions within the XML document or from different XML documents. For example, one <name>0</name> node might refer to the logical unit of "ge-0/0/0", while a second <name>0</name> node might refer to the logical unit of "lo0". Both of the two nodes are considered unique, even though their name and string-values are identical, but two references to the <name>0</name> node, both referring to the logical unit of "ge-0/0/0", could not both be present in the same node-set.

The union operator comes from XPath, so there are no restrictions on its use.

Example

This op script demonstrates the union operator.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns dyn = "http://exslt.org/dynamic";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get fe nodes */
        var $fe-rpc = {
            <get-interface-information> {
                <terse>;
                <interface-name> "fe*";
            }
        }
        var $fe-nodes = jcs:invoke( $fe-rpc );

        /* Get ge nodes */
        var $ge-rpc = {
            <get-interface-information> {
                <terse>;
                <interface-name> "ge*";
            }
        }
        var $ge-nodes = jcs:invoke( $ge-rpc );

        /* Get choice of what to view */
        var $choice = jcs:get-input( "View 'fe' or 'ge' interfaces or 'both': " );

        /* Build expression based on choice */
        var $expression = {
            if( $choice == "fe" ) {
                expr "$fe-nodes";
            }
            else if( $choice == "ge" ) {
                expr "$ge-nodes";
            }
            else if( $choice == "both" ) {
                /* Use union of both $fe-nodes and $ge-nodes */
                expr "$fe-nodes | $ge-nodes";
            }
        }

        /* Process expression dynamically */
        var $nodes = dyn:evaluate( $expression );

        for-each( $nodes/physical-interface ) {
            <output> "Interface: " _ name;
            <output> "Status: " _ admin-status _ "/" _ oper-status;
        }
    }
```

```
}
```

Output

```
jnpr@srx210> op union
View 'fe' or 'ge' interfaces or 'both': ge
Interface: ge-0/0/0
Status: up/up
Interface: ge-0/0/1
Status: up/down

jnpr@srx210> op union
View 'fe' or 'ge' interfaces or 'both': both
Interface: fe-0/0/2
Status: up/down
Interface: fe-0/0/3
Status: up/down
Interface: fe-0/0/4
Status: up/down
Interface: fe-0/0/5
Status: up/down
Interface: fe-0/0/6
Status: up/down
Interface: fe-0/0/7
Status: up/down
Interface: ge-0/0/0
Status: up/up
Interface: ge-0/0/1
Status: up/down
```

Part 3: Functions

This section documents all of the functions that are available natively in SLAX 1.0. These functions come from a variety of sources including XPath, XSLT, EXSLT, and Junos itself. The functions are organized alphabetically based on their common namespace prefix, if one exists.

Any functions that have a namespace prefix must have their namespace defined within the script file, but the prefix used for the function does not have to match common one; however, it is highly recommended to do so in order to avoid confusion.

The function syntax is displayed with the arguments and return values specified by data type because, although SLAX often converts data types from one type to another automatically, this is not always possible, and the documentation of an argument's data type often provides insight into the purpose of the argument; however, if the data type is not sufficient by itself to describe the argument's purpose, then a clarifying description is included for the argument within square brackets. An asterisk is appended to a function argument in the documented syntax when that argument can occur a variable number of times. (e.g. the **concat**() function accepts a variable number of string arguments, so its syntax reflects this by appending a * to its final string argument).

boolean()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **boolean**(object)

Description

The **boolean**() function returns the boolean value of the provided argument. Data types are converted to boolean values in the following ways:

- Number – Zero or "Not a Number" is false. Any other number is true.
- String – Empty strings are false. Strings with one or more characters are true.
- Node-set – Empty node-sets are false. Node-sets with one or more nodes are true.
- Result tree fragment – Treated as a node-set with an implicit root node, so result tree fragments are always true whether they are empty or not.

Example

The following op script shows various examples of the **boolean**() function returning the boolean value of different data types.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $boolean1 = boolean( 1 );
    var $boolean2 = boolean( 0 );
    var $boolean3 = boolean( empty-location-path );
    var $boolean4 = boolean( $user );
    var $rtf = {
        <no-text>;
    }
    var $boolean5 = boolean( $rtf );

    expr jcs:output( "Boolean value of 1: ", $boolean1 );
    expr jcs:output( "Boolean value of 0: ",$boolean2 );
    expr jcs:output( "Boolean value of empty node-set: ", $boolean3 );
    expr jcs:output( "Boolean value of ", $user, ": ", $boolean4 );
    expr jcs:output( "Boolean value of rtf: ", $boolean5 );
}
```

Output

```
Boolean value of 1: true
Boolean value of 0: false
Boolean value of empty node-set: false
Boolean value of lab: true
Boolean value of rtf: true
```

ceiling()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **ceiling**(number)

Description

The **ceiling**() function is one of the available SLAX rounding functions. It rounds the number argument up to return the smallest integer that is not less than the argument.

Example

This example shows the result of the **ceiling**() function with many different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> ceiling( 1 );
        <output> ceiling( 0.9 );
        <output> ceiling( 5.1 );
        <output> ceiling( 5.5 );
        <output> ceiling( 5.999999999 );
        <output> ceiling( -3.9 );
        <output> ceiling( -0.9 );
        <output> ceiling( -3.1 );
    }
}
```

Output

```
 1
 1
 6
 6
 6
-3
 0
-3
```

concat()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **concat**(string, string)

string **concat**(string, string, string*)

Description

The **concat**() function is used to combine two or more strings together. A minimum of two string arguments are required but more can be provided if additional strings should be included in the final concatenated result. If any of the provided arguments are not strings then they will be converted to strings using the same process as the **string**() function before being concatenated.

The returned result is the concatenated string of all the arguments in their sequential order.

Example

This op script example shows how to use the **concat**() function to combine multiple strings together.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $user-name = "lab";
        var $connections = 5;
```

```
        <output> concat( "User ", $user-name, " has connected ", $connections, " times." );
    }
}
```

Output

```
User lab has connected 5 times.
```

contains()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **contains**(string *[target]*, string *[substring]*)

Description

The **contains**() function tests if a string contains a second string. If the second string argument can be found within the first string argument then the function will return true. Otherwise, the function will return false.

Example

This op script example shows how to use the **contains**() function to only display lines that have a specific string within them. In this example, only the "cscript" processes will be output to the screen.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $rpc = <command> "show system processes extensive";

    var $results = jcs:invoke( $rpc );

    /* Break up into lines */
    var $lines = jcs:break-lines( $results );

    /* Print lines that contain cscript */
    for-each( $lines[contains( ., "cscript" )] ) {
        expr jcs:output( . );
    }
}
```

jcs:invoke() results:

```
<output>
    last pid: 47138;  load averages:  0.02,  0.06,  0.05  up 54+03:07:34    09:26:47
    110 processes: 2 running, 88 sleeping, 20 waiting

    Mem: 88M Active, 108M Inact, 48M Wired, 71M Cache, 110M Buf, 668M Free
    Swap:


      PID USERNAME  THR PRI NICE   SIZE    RES STATE    TIME   WCPU COMMAND
       11 root        1 171   52     0K    16K RUN    1148.0 92.29% idle
      581 root        1   4    0 11860K  2876K kqread  78.7H  3.03% chassism
    47111 root        1   4    0  4984K  2708K sbwait   0:00  3.00% cscript
      723 root        1   8    0 63904K  6692K nanslp  41.7H  0.05% pfem
       13 root        1 -20 -139     0K    16K WAIT   368:02  0.00% swi7: clock
    27746 root        2  44  -52 62232K  6292K select 170:38  0.00% sfid
...cut for brevity...
</output>
```

Output

```
47111 root        1   4    0  4984K  2708K sbwait   0:00  3.00% cscript
```

count()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **count**(node-set)

Description

The **count**() function returns the number of nodes contained within a node-set.

Example

This op script example shows various usages of the **count**() function. It is used to retrieve the number of nodes in a node-set variable as well as the number of nodes returned by different location paths.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    var $interface-rpc = {
        <get-interface-information> {
            <descriptions>;
        }
    }

    var $interfaces = jcs:invoke( $interface-rpc );

    var $top-level-count = count( $interfaces );
    expr jcs:output( "# of <interface-information>: ", $top-level-count );

    var $interface-count = count( $interfaces/physical-interface );
    expr jcs:output( "# of <physical-interface>: ", $interface-count );

    var $up-count = count( $interfaces/physical-interface[oper-status == "up"] );
    expr jcs:output( "# of up interfaces: ", $up-count );
```

jcs:invoke() results:

```
<interface-information junos:style="description">
    <physical-interface>
        <name>ge-0/3/0</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <description>Customer A</description>
    </physical-interface>
    <physical-interface>
        <name>fxp0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <description>OOB Management</description>
    </physical-interface>
</interface-information>
```

Output

```
# of <interface-information>: 1
# of <physical-interface>: 2
# of up interfaces: 1
```

current()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

node-set **current**()

Description

The **current**() function returns a node-set with a single member: the current node. Typically, the current node is identical to the context node; however, they diverge within predicates and after the first step of a location path, so the **current**() function is typically only used in those scenarios.

Example

This op script example shows how the **current**() function can be used to differentiate between the name child element of the context node and the name child element of the current node.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $configuration = jcs:invoke( "get-configuration" );

    for-each( $configuration/firewall/family/inet/filter ) {

        /* Determine if the filter has a matching term or not */
        if( term[ name == current()/name ] ) {
            expr jcs:output( "Filter ", name, " has a matching term." );
        }
        else {
            expr jcs:output( "Filter ", name, " has no matching term." );
        }
    }
}
```

Firewall configuration:

```
firewall {
    family inet {
        filter match {
            term match {
                then accept;
            }
        }
        filter no-match {
            term accept {
                then accept;
            }
        }
    }
    filter test {
        term test {
            then accept;
        }
    }
}
```

Output

```
Filter match has a matching term.
Filter no-match has no matching term.
```

date:add()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[date]* **date:add**(string *[date]*, string *[duration]*)

Description

The **date:add**() function adds a duration to a date value and returns the result. The first argument is a string in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gYear

The second argument is a string duration expressed in xs:duration format.

The return value is a string in one of the first argument's supported date formats. It is returned in the same format as the first argument unless a more precise format is required. For example, if months are added to a xs:gYear than it will likely be necessary to return the value as a xs:gYearMonth.

Fields that are not expressed in a particular format are set to their initial value, so the day and month are 1 and the time of day is 00:00:00 when they are not included in a format. This means that adding a duration of "P5D" (five days) to a date of "2001-10" would return a value of "2001-10-6".

If the timezone of the date value is not specified then it is assumed to be UTC, but if the timezone is specified to be something other than UTC then it is converted as part of the add operation. The return timezone is always UTC, although the "Z" timezone field is not always included in the return value unless the return value is in xs:dateTime format.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how to durations to date strings in multiple different formats by using the **date:add**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Add 4 months and 15 days */
        <output> date:add( "2010", "P4M15D" );

        /* Subtract 1 year */
        <output> date:add( "2001-01-01T00:00:00", "-P1Y" );

        /* Add 1 month and 2 days */
        <output> date:add( "2010-03", "P1M2D" );

        /* Add 1 year, 2 months, 3 days, 4 hours, 5 minutes, and 6 seconds */
        <output> date:add( "2001-01-01", "P1Y2M3DT4H5M6S" );

        /* Add 1 hour, 30 minutes */
        <output> date:add( "2001-01-01T01:30:00-07:00", "PT1H30M" );
    }
}
```

Output

```
2010-05-16
2000-01-01T00:00:00Z
2010-04-03
2002-03-04T04:05:06Z
2001-01-01T10:00:00Z
```

date:add-duration()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[duration]* **date:add-duration**(string *[duration]*, string *[duration]*)

Description

The **date:add-duration**() function combines two duration values together and returns the result. Both arguments are expressed in xs:duration format, and the return value is in xs:duration format as well.

An empty string is returned if either of the two strings is not valid, or if the two durations cannot be combined together. For example, if one duration is positive and the other is negative then the negative duration would have to be subtracted from the positive duration in order to determine the combined result; however, due to the variable number of days in a month it is not always possible to do this. Consider the durations of P1M and –P10D. These cannot be combined together without first exchanging the month value for its day quantity, but a month does not contain a consistent number of days, the amount differs based on the specific month in question, and durations have no concept of the specific dates, they are only generic references of a time period. Therefore, it isn't possible to combine these two durations together, and attempting to do so will result in an empty string being returned.

Note: See Appendix A for a description of the xs:duration format.

Example

This op script demonstrates how to add durations together with the **date:add-duration**() function, and it also shows an example of an invalid combination that results in a blank string being returned.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Add 1 day and 6 days */
        <output> date:add-duration( "P1D", "P6D" );

        /* Add 5 Months and 20 days to 15 days and 10 hours */
        <output> date:add-duration( "P5M20D", "P15DT10H" );

        /* Add 1 month and 1 minute */
        <output> date:add-duration( "P1M", "PT1M" );

        /* Add 10 years and negative five years */
        <output> date:add-duration( "P10Y", "-P5Y" );

        /* Example of invalid combination: add 2 months and negative 5 hours */
        <output> date:add-duration( "P2M", "-PT5H" );

        <output> "Completed";
    }
}
```

Output

```
P7D
P5M35DT10H
P1MT1M
P5Y
Completed
```

date:date()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[date]* **date:date**()

string *[date]* **date:date**(string *[date]*)

Description

The **date:date**() function returns a date specified by the string argument, or if no argument is provided it returns the current system date. The string argument must be in either xs:dateTime or xs:date format. The returned string is in xs:date format.

If an invalid string argument is provided then the returned string is blank.

If the date's time-zone is not UTC then the returned date string will include a reference to the time-zone, but dates in UTC time often omit the UTC time-zone indicator.

Note: See Appendix A for a description of the various date formats.

Example

This op script shows how **date:date**() can be used to return the date string for a number of different input values.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date();

        /* Date provided in xs:dateTime format */
        <output> date:date( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:date( "2001-01-01" );

        /* Date provided in xs:dateTime format with included time-zone */
```

```
        <output> date:date( "2010-10-05T01:30:00-07:00" );

        /* xs:gYearMonth is unsupported, so it will return a blank string */
        <output> date:date( "2010-03" );

        <output> "Completed";
    }
}
```

Output

```
2010-10-14
2001-02-03
2001-01-01
2010-10-05-07:00
Completed
```

date:date-time()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[date-time]* **date:date-time**()

Description

The **date:date-time**() function returns the current system date and time, expressed in xs:dateTime format with included time-zone.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates the use of **date:date-time**() to display the current date and time. A multi-second pause is introduced into the script so that the two calls to **date:date-time**() return different results.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
```

```
    <op-script-results> {

        /* Display todays date */
        <output> "Starting date and time is " _ date:date-time();

        /* Pause for five seconds */
        expr jcs:sleep(5);

        /* Display the time again */
        <output> "Ending date and time is " _ date:date-time();

    }
}
```

Output

```
Starting date and time is 2010-10-15T23:33:23Z
Ending date and time is 2010-10-15T23:33:28Z
```

date:day-abbreviation()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[abbreviation]* **date:day-abbreviation**()

string *[abbreviation]* **date:day-abbreviation**(string *[date]*)

Description

The **date:day-abbreviation**() function returns the three letter abbreviation of a date's day of the week: "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", or "Sat". A string argument can be provided, in xs:date or xs:dateTime format, to indicate the date for which the abbreviation should be returned. If no argument is provided then the current date is used. A string argument in an invalid format results in an empty string being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:day-abbreviation**() can be used to return the abbreviation of both the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
```

```
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:day-abbreviation();

        /* Date provided in xs:dateTime format */
        <output> date:day-abbreviation( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:day-abbreviation( "2001-01-01" );

        /* Date provided in xs:dateTime format with included time-zone */
        <output> date:day-abbreviation( "2010-10-05T01:30:00-07:00" );

        /* xs:gYearMonth is unsupported, so it will return a blank string */
        <output> date:day-abbreviation( "2010-03" );

        <output> "Completed";
    }
}
```

Output

```
2010-10-15 = Fri
Sat
Mon
Tue
Completed
```

date:day-in-month()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[day]* **date:day-in-month**()

number *[day]* **date:day-in-month**(string *[date]*)

Description

The **date:day-in-month**() function returns the date's day as a number. For example, October 15th would be returned as the number 15. A string argument can be provided to indicate the date to use. The valid formats for this argument are:

- xs:dateTime
- xs:date
- xs:gMonthDay
- xs:gDay

If no argument is provided then the current date is returned. If the argument does not follow a valid format then NaN is returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:day-in-month**() can be used to return the day number of the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:day-in-month();

        /* Date provided in xs:dateTime format */
        <output> date:day-in-month( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:day-in-month( "2001-01-01" );

        /* Date provided in xs:gMonthDay format */
        <output> date:day-in-month( "--05-31" );

        /* Date provided in xs:gDay format */
        <output> date:day-in-month( "---26" );

    }
}
```

Output

```
2010-10-16 = 16
3
1
31
26
```

date:day-in-week()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[day]* **date:day-in-week**()

number *[day]* **date:day-in-week**(string *[date]*)

Description

The **date:day-in-week**() function returns the number value of a date's day of the week: Sunday = 1, Monday = 2, Tuesday = 3, Wednesday = 4, Thursday = 5, Friday = 6, and Saturday = 7. A string argument can be provided, in xs:date or xs:dateTime format, to indicate the date for which the day in week number should be returned. If no argument is provided then the current date is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:day-in-week**() can be used to return the day of the week value of both the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:day-in-week();

        /* Date provided in xs:dateTime format */
        <output> date:day-in-week( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:day-in-week( "2001-01-01" );

    }
}
```

Output

```
2010-10-16 = 7
7
2
```

date:day-in-year()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[day]* **date:day-in-year**()

number *[day]* **date:day-in-year**(string *[date]*)

Description

The **date:day-in-year**() function returns the number value of a date's day of the year. A string argument can be provided, in xs:date or xs:dateTime format, to indicate the date for which the day in year number should be returned. If no argument is provided then the current date is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:day-in-year**() can be used to return the day of the year value of both the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:day-in-year();

        /* Date provided in xs:dateTime format */
        <output> date:day-in-year( "2001-02-03T05:00:00");
```

```
        /* Date provided in xs:date format */
        <output> date:day-in-year( "2001-01-01" );

    }
}
```

Output

```
2010-10-16 = 289
34
1
```

date:day-name()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[name]* **date:day-name**()

string *[name]* **date:day-name**(string *[date]*)

Description

The **date:day-name**() function returns the name of a date's day of the week: "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", or "Saturday". A string argument can be provided, in xs:date or xs:dateTime format, to indicate the date for which the day name should be returned. If no argument is provided then the current date is used. A string argument in an invalid format results in an empty string being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:day-name**() can be used to return both the name of the current date as well as the provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";
```

```
match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:day-name();

        /* Date provided in xs:dateTime format */
        <output> date:day-name( "2001-02-04T05:00:00");

        /* Date provided in xs:date format */
        <output> date:day-name( "2001-01-01" );

        /* Date provided in xs:dateTime format with included time-zone */
        <output> date:day-name( "2010-10-05T01:30:00-07:00" );

    }
}
```

Output

```
2010-10-16 = Saturday
Sunday
Monday
Tuesday
```

date:day-of-week-in-month()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[week]* **date:day-of-week-in-month**()

number *[week]* **date:day-of-week-in-month**(string *[date]*)

Description

The **date:day-of-week-in-month**() function returns the day-of-week number within a month. For example, the 2nd Sunday of the month returns 2, and the 3rd Saturday of the month returns 3. A string argument can be provided, in xs:date or xs:dateTime format, to indicate the date for which a value should be returned. If no argument is provided then the current date is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates multiple examples of the **date:day-of-week-in-month**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " is " _ date:day-name() _ " # " _
            date:day-of-week-in-month() _ " of month";

        /* Date provided in xs:date format */
        var $date = "2010-01-01";
        <output> date:date( $date ) _ " is " _ date:day-name( $date ) _ " # " _
            date:day-of-week-in-month( $date ) _ " of month";


    }
}
```

Output

```
2010-10-16 is Saturday # 3 of month
2010-01-01 is Friday # 1 of month
```

date:difference()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[duration]* **date:difference**(string *[date]*, string *[date]*)

Description

The **date:difference**() function returns the difference between the two string arguments as a string in duration format. The valid formats for the arguments are:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gYear

If the first date occurs prior to the second date, then the returned duration will be positive, otherwise it will be negative.

If dates of different formats are compared then the more precise date is truncated to match the least precise date. For example, if the function is checking the difference between a xs:gYear of "1980" and a xs:gYearMonth of "1985-05", then it will first truncate the xs:gYearMonth to be a xs:gYear of "1985", and then it will return a duration of 5 years: "P5Y".

When computing the difference at xs:gYear precision, the duration is expressed in years only, but when computing the difference at xs:gYearMonth precision, the duration can be expressed in years and/or months, with the number of months always being less than 12.

When computing the difference at xs:date precision, the duration is expressed in days only. This is because durations are generic time periods, so there is no way to convert from days into months or years because each of those latter units have differing numbers of days based on the month or whether or not a year is a leap year. When computing the difference at xs:dateTime precision, the duration can be expressed in days, hours, minutes, and/or seconds, with the number of seconds and minutes always being less than 60, and the number of hours always being less than 24.

When no difference is present, the returned duration is "P0D", even if the difference is being computed at xs:gYear or xs:gYearMonth precision.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates the types of results returned by the **date:difference**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* If one is a xs:gYear, then the comparison is between years */
        <output> "xs:gYear precision comparisons:";
        <output> date:difference( "2001", "2005" );
        <output> date:difference( "2009-12", "2010" );

        /* Or, if one is a xs:gYearMonth, then the comparison is between months */
        <output> "xs:gYearMonth precision comparisons:";
        <output> date:difference( "2010-09", "2010-12" );
        <output> date:difference( "2010-09", "2010-10-01" );

        /* Or, if one is a xs:date then the comparison is between days */
        <output> "xs:date precision comparisons:";
        <output> date:difference( "2010-01-01", "2010-02-01" );
        <output> date:difference( "2010-01-01", "2010-03-01T10:00:00" );

        /* If both are xs:dateTime then compare between days, hours, minutes, and seconds */
```

```
        <output> "xs:dateTime precision comparisons:";
        <output> date:difference( "2010-10-18T12:00:00", "2010-10-19T12:12:34" );
        <output> date:difference( "2010-10-31T00:00:00", "2010-10-31T01:01:01" );

        /* Zero duration is expressed as P0D */
        <output> "Zero duration examples:";
        <output> date:difference( "2001", "2001-01" );
        <output> date:difference( "2010-10-18", "2010-10-18" );
        <output> date:difference( "2010-10-19T05:00:00", "2010-10-19T05:00:00" );
    }
}
```

Output

```
xs:gYear precision comparisons:
P4Y
P1Y
xs:gYearMonth precision comparisons:
P3M
P1M
xs:date precision comparisons:
P31D
P59D
xs:dateTime precision comparisons:
P1DT12M34S
PT1H1M1S
Zero duration examples:
P0D
P0D
P0D
```

date:duration()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[duration]* **date:duration**()

string *[duration]* **date:duration**(number *[seconds]*)

Description

The **date:duration**() function returns the duration string for the number of seconds provided in the functions argument. If no argument is provided then the function returns the duration for the number of seconds from 1970-01-01T00:00:00Z to the current date/time. The duration will be expressed in days, hours, minutes, and seconds only, with the number of hours always less than 24 and the number of minutes and seconds always less than 60. A duration of zero is expressed as “P0D”. If the number is NaN or Infinity then a blank string is returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates the types of results returned by the **date:duration**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> "Current time: " _ date:duration();
        <output> "0s duration: " _ date:duration( 0 );
        <output> "1s duration: " _ date:duration( 1 );
        <output> "60s duration: " _ date:duration( 60 );
        <output> "3600s duration: " _ date:duration( 3600 );
        <output> "5000s duration: " _ date:duration( 5000 );
        <output> "86,400s duration: " _ date:duration( 86400 );
        <output> "10,000,000s duration: " _ date:duration( 10000000 );
        <output> "NaN duration: " _ date:duration( number( "NaN" ) );
    }
}
```

Output

```
Current time: P14900DT22H42M34S
0s duration: P0D
1s duration: PT1S
60s duration: PT1M
3600s duration: PT1H
5000s duration: PT1H23M20S
86,400s duration: P1D
10,000,000s duration: P115DT17H46M40S
NaN duration:
```

date:hour-in-day()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[hour]* **date:hour-in-day**()

number *[hour]* **date:hour-in-day**(string *[date]*)

Description

The **date:hour-in-day**() function returns the hour number, in 24-hour format, from a date/time value. A string argument can be provided, in xs:time or xs:dateTime format, to indicate the date/time for which the hour number should be returned. If no argument is provided then the current date/time is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates multiple uses of the **date:hour-in-day**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display current hour */
        <output> date:date-time() _ " = " _ date:hour-in-day();

        /* Date provided in xs:dateTime format */
        <output> date:hour-in-day( "2001-02-03T05:00:00");

        /* Date provided in xs:time format */
        <output> date:hour-in-day( "23:58:59" );

    }
}
```

Output

```
2010-10-19T16:01:43Z = 16
5
23
```

date:leap-year()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

boolean **date:leap-year**()

boolean **date:leap-year**(string *[date]*)

Description

The **date:leap-year**() function returns true if the date provided falls within a leap year or false if it does not. The date can be provided as a string in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gYear

If no argument is provided then the current date/time is used. If an invalid format is used then NaN is returned, which means that the returned object type in that case is actually a number rather than a boolean. (A number with a value of NaN converts to a boolean value of false).

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates the return value of the **date:leap-year**() function based on a number of different years.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Create years structure to loop through */
        var $years := {
            <year> "2000";
            <year> "2001";
            <year> "2002";
            <year> "2003";
            <year> "2004";
            <year> "2005";
            <year> "2006";
            <year> "2007";
            <year> "2008";
            <year> "2009";
            <year> "2010";
        }
```

```
        /* Loop through and display if they are leap years or not */
        for-each( $years/year ) {
            if( date:leap-year( . ) ) {
                expr jcs:output( ., " is a leap year" );
            }
            else {
                expr jcs:output( . );
            }
        }

        /* Display current date */
        var $current-date = date:date();
        if( date:leap-year( $current-date ) ) {
            expr jcs:output( $current-date, " is in a leap year" );
        }
        else {
            expr jcs:output( $current-date, " is not in a leap year" );
        }

    }
}
```

Output

```
2000 is a leap year
2001
2002
2003
2004 is a leap year
2005
2006
2007
2008 is a leap year
2009
2010
2010-10-19 is not in a leap year
```

date:minute-in-hour()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[minute]* **date:minute-in-hour**()

number *[minute]* **date:minute-in-hour**(string *[time]*)

Description

The **date:minute-in-hour**() function returns the minute of the hour number from a date/time value. A string

argument can be provided, in xs:time or xs:dateTime format, to indicate the date/time for which the minute number should be returned. If no argument is provided then the current date/time is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates multiple uses of the **date:minute-in-hour**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display current minute */
        <output> date:date-time() _ " = " _ date:minute-in-hour();

        /* Date provided in xs:dateTime format */
        <output> date:minute-in-hour( "2001-02-03T05:00:00");

        /* Date provided in xs:time format */
        <output> date:minute-in-hour( "23:58:59" );

    }
}
```

Output

```
2010-10-19T16:20:55Z = 20
0
58
```

date:month-abbreviation()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[abbreviation]* **date:month-abbreviation**()

string *[abbreviation]* **date:month-abbreviation**(string *[date]*)

Description

The **date:month-abbreviation**() function returns the three letter abbreviation of a date's month: "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", or "Dec". A string argument can be provided to indicate the date for which the month abbreviation should be returned. It must be in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gMonth
- xs:gMonthDay

If no argument is provided then the current date is used. A string argument in an invalid format results in an empty string being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:month-abbreviation**() can be used to return the month abbreviation of both the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:month-abbreviation();

        /* Date provided in xs:dateTime format */
        <output> date:month-abbreviation( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:month-abbreviation( "2001-01-01" );

        /* Date provided in xs:gYearMonth format */
        <output> date:month-abbreviation( "2010-03" );

        /* Date provided in xs:gMonth format */
        <output> date:month-abbreviation( "--05--" );

        /* Date provided in xs:gMonthDay format */
```

```
            <output> date:month-abbreviation( "--06-01" );

        }
    }
```

Output

```
2010-10-21 = Oct
Feb
Jan
Mar
May
Jun
```

date:month-in-year()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[month]* **date:month-in-year**()

number *[month]* **date:month-in-year**(string *[date]*)

Description

The **date:month-in-year**() function returns the number value of a date's month, where January = 1, February = 2, March = 3, April = 4, May = 5, June = 6, July = 7, August = 8, September = 9, October = 10, November = 11, and December = 12. A string argument can be provided to indicate the date for which the month's number value should be returned. It must be in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gMonth
- xs:gMonthDay

If no argument is provided then the current date is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:month-in-year**() can be used to return the month's number value of both the current date as well as provided dates in various formats.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:month-in-year();

        /* Date provided in xs:dateTime format */
        <output> date:month-in-year( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:month-in-year( "2001-01-01" );

        /* Date provided in xs:gYearMonth format */
        <output> date:month-in-year( "2010-03" );

        /* Date provided in xs:gMonth format */
        <output> date:month-in-year( "--05--" );

        /* Date provided in xs:gMonthDay format */
        <output> date:month-in-year( "--06-01" );

    }
}
```

Output

```
2010-10-21 = 10
2
1
3
5
6
```

date:month-name()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[name]* **date:month-name**()

string *[name]* **date:month-name**(string *[date]*)

Description

The **date:month-name**() function returns the name of a date's month. A string argument can be provided to indicate the date for which the month name should be returned. It must be in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gMonth
- xs:gMonthDay

If no argument is provided then the current date is used. A string argument in an invalid format results in an empty string being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:month-name**() can be used to return the month name of both the current date as well as provided dates in a variety of formats.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:month-name();

        /* Date provided in xs:dateTime format */
        <output> date:month-name( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:month-name( "2001-01-01" );

        /* Date provided in xs:gYearMonth format */
        <output> date:month-name( "2010-03" );

        /* Date provided in xs:gMonth format */
        <output> date:month-name( "--05--" );

        /* Date provided in xs:gMonthDay format */
        <output> date:month-name( "--06-01" );

    }
}
```

Output

```
2010-10-21 = October
February
January
March
May
June
```

date:second-in-minute()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[second]* **date:second-in-minute**()

number *[second]* **date:second-in-minute**(string *[time]*)

Description

The **date:second-in-minute**() function returns the second of the minute number from a date/time value. A string argument can be provided, in xs:time or xs:dateTime format, to indicate the date/time for which the second number should be returned. If no argument is provided then the current date/time is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates multiple uses of the **date:second-in-minute**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display current second */
        <output> date:date-time() _ " = " _ date:second-in-minute();
```

```
            /* Date provided in xs:dateTime format */
            <output> date:second-in-minute( "2001-02-03T05:00:00");

            /* Date provided in xs:time format */
            <output> date:second-in-minute( "23:58:59" );

        }
    }
```

Output

```
2010-10-21T21:44:03Z = 3
0
59
```

date:seconds()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[seconds]* **date:seconds**()

number *[seconds]* **date:seconds**(string *[date or duration]*)

Description

The **date:seconds**() function returns the number of seconds specified by either a date or duration value. If the value is a date then the result is the number of seconds since 1970-01-01T00:00:00Z, so earlier dates return a negative value and later dates return a positive value. The following formats are accepted for the string argument:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gYear
- xs:duration

If a duration is used rather than a date then the year and month values of the duration must be zero. If no argument is provided then the current date/time is used. If an invalid format is provided then NaN is returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates multiple uses of the **date:seconds**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display current seconds */
        <output> date:date-time() _ " = " _ date:seconds();

        /* Before 1970-01-01T00:00:00Z */
        <output> date:seconds( "1969-12-31T00:00:00Z");

        /* On 1970-01-01T00:00:00Z */
        <output> date:seconds( "1970-01-01T00:00:00Z");

        /* After 1970-01-01T00:00:00Z */
        <output> date:seconds( "1970-01-01T00:00:30Z");

        /* In xs:gYearMonth format */
        <output> date:seconds( "1970-02" );

        /* Duration value */
        <output> date:seconds( "PT2M" );

        /* Negative duration value */
        <output> date:seconds( "-PT1H" );
    }
}
```

Output

```
2010-10-21T22:23:42Z = 1287699822
-86400
0
30
2678400
120
-3600
```

date:sum()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[duration]* **date:sum**(node-set *[durations]*)

Description

The **date:sum**() function combines a node-set of duration values together and returns the result. All of the nodes within the node-set must be expressed in xs:duration format, and the return value is in xs:duration format as well.

An empty string is returned if any of the node values is not valid, or if the durations cannot be combined together. For example, if one duration is positive and another is negative then the negative duration would have to be subtracted from the positive duration in order to determine the combined result; however, due to the variable number of days in a month it is not always possible to do this. Consider the durations of P1M and –P10D. These cannot be combined together without first exchanging the month value for its day quantity, but a month does not contain a consistent number of days, the amount differs based on the specific month in question, and durations have no concept of the specific dates; they are only generic references of a time period. Therefore, it isn't possible to combine these two durations together and attempting to do so will result in an empty string being returned.

Note: See Appendix A for a description of the xs:duration format.

Example

This op script demonstrates how to add durations together with the **date:sum**() function, and it also shows an example of an invalid combination that results in a blank string being returned.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Add time values together */
        var $set1 := {
            <duration> "PT1H";
            <duration> "PT1M";
            <duration> "PT1S";
        }
        <output> date:sum( $set1/duration );

        /* Add date values together */
        var $set2 := {
            <duration> "P10Y";
            <duration> "P5M";
            <duration> "P4D";
        }
        <output> date:sum( $set2/duration );
```

```
            /* Example of a set that cannot be combined */
            var $set3 := {
                <duration> "P5Y";
                <duration> "P1M";
                <duration> "-P2D";
            }
            <output> date:sum( $set3/duration );

            /* Add dates and time values together */
            var $set4 := {
                <duration> "P5YT5S";
                <duration> "P3MT10H";
                <duration> "-PT5H";
            }
            <output> date:sum( $set4/duration );

        }
    }
```

Output

```
PT1H1M1S
P10Y5M4D
P5Y3MT5H5
```

date:time()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

string *[time]* **date:time**()

string *[time]* **date:time**(string *[time]*)

Description

The **date:time**() function returns a time specified by the string argument, or if no argument is provided it returns the current system time. The string argument must be in either xs:dateTime or xs:time format. The returned string is in xs:time format.

If an invalid string argument is provided then the returned string is blank. If the date's time-zone is not UTC then the returned date string will include a reference to the time-zone, but dates in UTC time often omit the UTC time-zone indicator.

Note: See Appendix A for a description of the various date formats.

Example

This op script shows how **date:time**() can be used to return the time string for a number of different input values.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Current */
        <output> date:date-time() _ " = " _ date:time();

        /* From date */
        <output> date:time("2010-10-31T05:05:05");

        /* With time-zone */
        <output> date:time("2001-01-01T12:00:00-05:00");

        /* UTC */
        <output> date:time("2005-05-13T05:00:00+00:00");
    }
}
```

Output

```
2010-10-21T15:58:14-07:00 = 15:58:14-07:00
05:05:05
12:00:00-05:00
05:00:00
```

date:week-in-month()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[week]* **date:week-in-month**()

number *[week]* **date:week-in-month**(string *[date]*)

Description

The **date:week-in-month**() function returns the date's week within a month as a number. For example, October 15th, 2010 would be returned as the number 3. A string argument can be provided to indicate the date to use. The valid formats for this argument are:

- xs:dateTime
- xs:date

The first day of the month is within the first week, and week numbers change on Monday. So, if October 1st is Friday, then October 1st (Friday), 2nd (Saturday), and 3rd (Sunday) are in week 1, and October 4th (Monday) is in week 2.

If no argument is provided then the current date is used. If the argument does not follow a valid format then NaN is returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date:week-in-month**() can be used to return the week number of the current date as well as provided dates.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Current */
        <output> date:date-time() _ " = " _ date:week-in-month();

        /* xs:dateTime */
        <output> date:week-in-month("2010-10-31T05:05:05");

        /* xs:date */
        <output> date:week-in-month("2010-10-03");

        /* xs:date */
        <output> date:week-in-month("2010-10-04");
    }
}
```

Output

```
2010-10-21T16:09:38-07:00 = 4
5
1
2
```

date:week-in-year()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[week]* **date:week-in-year**()

number *[week]* **date:week-in-year**(string *[date]*)

Description

The **date:week-in-year**() function returns the date's week within a year as a number. A string argument can be provided to indicate the date to use. The valid formats for this argument are:

- xs:dateTime
- xs:date

The returned week number follows the ISO 8601 convention, which states that the first Thursday of the year is in the first week of the year with the week number incrementing each Monday. This means that the first few days of January will fall into the last week of the prior year unless January 1st that year is on a Thursday. If no argument is provided then the current date is used. If the argument does not follow a valid format then NaN is returned.

Note: See Appendix A for a description of the various date formats.

Note: As of the time of this writing, this function returns an incorrect week number in many cases.

Example

This op script demonstrates how **date:week-in-year**() can be used.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* This date is not susceptible to the current bug */
        var $good-date = "2009-01-01";

        <output> $good-date _ " is a " _ date:day-name( $good-date ) _
            " and is in week #" _ date:week-in-year( $good-date );
```

```
    }
}
```

Output

```
2009-01-01 is a Thursday and is in week #1
```

date:year()

Source: EXSLT
Namespace: http://exslt.org/dates-and-times
Common Prefix: date
Minimum Version: Junos 9.4

Syntax

number *[year]* **date: year**()

number *[year]* **date: year**(string *[date]*)

Description

The **date: year**() function returns the number value of a date's year. A string argument can be provided to indicate the date for which the year should be returned. It must be in one of the following formats:

- xs:dateTime
- xs:date
- xs:gYearMonth
- xs:gYear

If no argument is provided then the current date is used. A string argument in an invalid format results in NaN being returned.

Note: See Appendix A for a description of the various date formats.

Example

This op script demonstrates how **date: year**() can be used to return the year value of both the current date as well as provided dates in various formats.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display todays date */
        <output> date:date() _ " = " _ date:year();

        /* Date provided in xs:dateTime format */
        <output> date:year( "2001-02-03T05:00:00");

        /* Date provided in xs:date format */
        <output> date:year( "2003-01-01" );

        /* Date provided in xs:gYearMonth format */
        <output> date:year( "2010-03" );

        /* Date provided in xs:gYear format */
        <output> date:year( "2012" );

    }
}
```

Output

```
2010-10-21-07:00 = 2010
2001
2003
2010
2012
```

document()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

node-set *[document]* **document**(string *[URI]*)

node-set *[document]* **document**(string *[URI]*, node-set *[modify default directory]*)

node-set *[documents]* **document**(node-set *[URIs]*)

node-set *[documents]* **document**(node-set *[URIs]*, node-set *[modify default directory]*)

Description

The **document**() function loads a XML document from local storage and returns the XML contents to the calling script as a node-set. The first argument provided to the function is a URI pointing to the local file that should be retrieved. If the argument is a string then a single file is returned, or if the argument is a node-set then each node is converted to a string and all of their contents are returned within a single node-set. The

returned node-set consists of a single root node to the retrieved XML document (or union of multiple documents if a node-set argument is used). Alternatively, a fragment identifier can be included within the URI, using XPointer syntax, to specify that only a portion of the document should be retrieved. In this scenario, rather than including a single root node within the returned node-set, the node-set will contain all the nodes within the XML document identified by the fragment identifier. As an example, the following code would retrieve all <email> nodes from within the example.xml file:

```
var $email-nodes = document( "/var/home/jnpr/example.xml#xpointer(//email)" );
```

Note that Junos does not strip namespaces from XML documents retrieved from the **document**() function. This differs from the XML documents returned by Junos in response to a JUNOS XML API request, and must be considered when crafting location paths to pull data from these node-sets.

The second **document**() argument is optional and only serves to change the default directory from which a relative URI is resolved. The default directory is determined in the following manner:

When a single argument is provided to **document**()**:**

- If the argument is a string then the location of the executing script (e.g. /var/run/scripts/op) is used as the default directory.
- If the argument is a node-set, and the node-set was not learned from the **document**() function, then the default directory is /var/tmp.
- If the argument is a node-set, and the node-set was learned from the **document**() function, then the default directory is the same directory as the file loaded by that function.

When two arguments are provided to **document**()**:**

- If the second argument node-set was retrieved through the **document**() function, then the default directory is the same directory as the file loaded by that function.
- If the second argument node-set was not retrieved through the **document**() function then the default directory is /var/tmp.
- If the second argument node-set is empty then the location of the executing script is used as the default directory.

If the file does not exist, or is not readable then an empty node-set is returned and an error message is displayed: "failed to load external entity".

One special usage of the **document**() function is to retrieve the contents of the current script by calling the function with an empty string as its argument: **document("");** however, because SLAX scripts are not valid XML files this usage is only applicable to XSLT scripts.

Calling the **document**() function to read the same file multiple times within a script will not reflect any changes made to that file in the interim because the file is cached when first read and the original contents are returned every time **document**() retrieves the same file.

Only files on the local hard disk can be read by the **document**() function. Remote files cannot be retrieved from FTP or HTTP servers or even from other routing-engines in the same chassis. Instead, the file would first need to be copied to the local hard disk and then read by the **document**() function.

Text files cannot be read by the **document**() function. It will display an error if the file is not in XML format. The <file-get> Junos API element can be used instead if a text file needs to be read by the script.

The **document**() function always accesses the file system as user "nobody", so it can only read files that grant read permission to user "nobody" or to everyone.

Example

This op script example shows how the **document**() function can retrieve the XML contents from an external file, allowing the script to use the returned data.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Load document into script */
        var $chassis-info = document("/var/home/jnpr/chassis-inventory.xml");

        /*
         * Retrieve chassis name, use local-name() function to avoid
         * dealing with namespaces
         */
        var $chassis = $chassis-info//*[local-name() == "chassis"];
        var $chassis-name = $chassis/*[local-name() == "name"];

        <output> "Chassis name is " _ $chassis-name;
    }
}
```

chassis-inventory.xml

```
<chassis-inventory xmlns="http://xml.juniper.net/junos/10.1R1/junos-chassis">
    <chassis>
        <name>Chassis</name>
        <serial-number>xxxxxxxxxxxx</serial-number>
        <description>SRX210-hm</description>
        <chassis-module>
            <name>Routing Engine</name>
            <version>REV 28</version>
            <part-number>750-021779</part-number>
            <serial-number>xxxxxxxx</serial-number>
            <description>RE-SRX210-HIGHMEM</description>
        </chassis-module>
        <chassis-module>
            <name>FPC 0</name>
            <description>FPC</description>
            <chassis-sub-module>
                <name>PIC 0</name>
                <description>2x GE, 6x FE, 1x 3G</description>
            </chassis-sub-module>
        </chassis-module>
        <chassis-module>
            <name>Power Supply 0</name>
        </chassis-module>
    </chassis>
</chassis-inventory>
```

Output

```
Chassis name is Chassis
```

dyn:evaluate()

Source: EXSLT
Namespace: http://exslt.org/dynamic
Common Prefix: dyn
Minimum Version: Junos 9.4

Syntax

object *[result]* **dyn:evaluate**(string *[XPath]*)

Description

The **dyn:evaluate**() function processes an XPath expression dynamically. It takes a single string argument, which is the expression that should be executed. This expression is evaluated as if the exact string were written in the code in place of the **dyn:evaluate**() function call. This means that the context information is the same as the location where the **dyn:evaluate**() function was called, including:

- Context node, size, and position
- Current node
- Variables and parameters
- Functions
- Namespace declarations
- Keys
- Decimal formats

If the string argument is blank then an empty node-set is returned. If the string argument is an invalid XPath expression then an error message is displayed and an empty node-set is returned.

The function returns the result of the expression, which could be in any of the standard data types.

When using this function, keep the following in mind:

- These SLAX-specific operators are not supported in the expression:
  - "==" equality operator
    - Must use "="
  - "&&" and operator
    - Must use "and"
  - "||" or operator
    - Must use "or"

- "_" string concatenating operator
  - Must use **concat**() function
- Functions can be called but templates cannot.
- Do not terminate the expression with a semicolon.
- Do not start the expression with the **expr** statement.
- The namespace of any functions used within the expression must be defined within the SLAX script.

Potential uses

- Calling a function that is determined dynamically, such as through user input.
- Calling a function with a dynamic number of arguments.
- Performing an operation or comparison where the operator is determined dynamically.
- Building location paths dynamically, such as through user input.

Note: The **dyn:evaluate**() and **saxon:evaluate**() functions have equivalent functionality.

Example

This op script demonstrates two example uses of **dyn:evaluate**(). The first example needs to use **dyn:evaluate**() because the function to be called is determined dynamically. The second example actually didn't need to use **dyn:evaluate**(), but it was included to point out that the equality operator needs to be "=" rather than "==". Also, notice that the necessary namespaces to run the expected functions have been defined in the script.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns dyn = "http://exslt.org/dynamic";
ns math = "http://exslt.org/math";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Ask for the function */
        var $prompt = "Choose an EXSLT math function\n" _
                      "(abs,acos,asin,atan,cos,exp,log,sin,sqrt,tan)\n" _
                      "Enter selection: ";
        var $function = jcs:get-input( $prompt );

        /* Ask for the number */
        var $number = jcs:get-input( "Enter number to compute: " );

        /* Retrieve the result */
        var $expression = "math:" _ $function _ "( $number )";
        expr jcs:output( "Expression used: ", $expression );
        expr jcs:output(  "The answer is ", dyn:evaluate( $expression ) );
```

```
            /* This example allows the user to select an index to tokenize */
            var $example := {
                <node> "one";
                <node> "two";
                <node> "three";
                <node> "four";
            }
            expr jcs:output( "_\r \nOptions:" );
            for-each( $example/node ) {
                expr jcs:output( . );
            }
            var $index = jcs:get-input( "Select index to tokenize (1-4): " );

            /* dyn:evaluate must use XSLT syntax, so equality operator is = instead of == */
            var $expression2 = "str:tokenize( $example/node[ position() = " _ $index _ "], \"\"
)";

            /* retrieve and display the tokenized string */
            var $tokens = dyn:evaluate( $expression2 );
            for-each( $tokens ) {
                expr jcs:output( . );
            }
        }
    }
```

Output

```
Choose an EXSLT math function
(abs,acos,asin,atan,cos,exp,log,sin,sqrt,tan)
Enter selection: abs
Enter number to compute: -5
Expression used: math:abs( $number )
The answer is 5
Options:
one
two
three
four
Select index to tokenize (1-4): 3
t
h
r
e
e
```

dyn:map()

Source: EXSLT
Namespace: http://exslt.org/dynamic
Common Prefix: dyn
Minimum Version: Junos 9.4

Syntax

node-set *[results]* **dyn:map**(node-set *[context nodes]*, string *[XPath]*)

Description

The **dyn:map**() function processes an XPath expression dynamically once for every node in the node-set argument and then returns each result as a node in the returned node-set. The second argument is a string expression that should be executed for every node in the first argument node-set. The expression is evaluated as if the exact string were written in the code in place of the **dyn:map**() function call. This means that the context information is the same as the location where the **dyn:map**() function was called, including:

- Current node
- Variables and parameters
- Functions
- Namespace declarations
- Keys
- Decimal formats

However, in contrast to the **dyn:evaluate**() function, the context node, size, and position are changed before each evaluation of the expression. The context node is set to the node being evaluated, the context size is set to the size of the node-set being evaluated, and the context position is set to the position of the node within its node-set.

If the string argument is blank then an empty node-set is returned. If the string argument is an invalid XPath expression then an error message is displayed and an empty node-set is returned.

If the return value of the expression is a boolean then an element node is added to the returned node-set with a name of <exsl:boolean> and a string value of "true" if the boolean is true and "" (blank) if the boolean is false.

If the return value of the expression is a string then an element node is added to the returned node-set with a name of <exsl:string> and a string value matching the string result.

If the return value of the expression is a number then an element node is added to the returned node-set with a name of <exsl:number> and a string value determined by converting the number value to a string.

If the return value of the expression is a node-set then it is combined with the existing nodes in the return node-set in the same manner as using the union operator.

If none of the returned values are node-sets then the position of the result node within the node-set will match the position of the original node in the argument; however, if any of the returned values are node-sets then the order of the returned node-set will often not match that of the node-set argument and the two node-sets could be a different size.

When using this function, keep the following in mind:

- These SLAX-specific operators are not supported in the expression:
  - "==" equality operator
    - Must use "="
  - "&&" and operator
    - Must use "and"
  - "||" or operator
    - Must use "or"

- “_” string concatenating operator
  - Must use **concat**() function
- Functions can be called but templates cannot.
- Do not terminate the expression with a semicolon.
- Do not start the expression with the **expr** statement.
- The namespace of any functions used within the expression must be defined within the SLAX script.

Example

This op script shows a simple example of the **dyn:evaluate**() function.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns dyn = "http://exslt.org/dynamic";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Create string node-set */
        var $ns := {
            <node> "example";
            <node> "beta";
            <node> "longer string";
        }

        /* output the combined string length */
        <output> "Combined length: " _ sum( dyn:map( $ns/node, "string-length( . )" ) );
    }
}
```

Output

```
Combined length: 24
```

element-available()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **element-available**(string *[name]*)

Description

The **element-available**() function returns true if the string argument matches the qualified name of a known instruction element. Qualified names are the namespace prefix and local name, separated by a colon. Example: xsl:sort, where xsl is the namespace prefix and sort is the local name. The namespace specified in the string argument provided to **element-available**() must be defined within the script.

Top-level elements are not considered instruction elements, so the **element-available**() function will return false when queried about elements such as <xsl:template> or <func:function>.

Unlike with functions, there is no way to add new extension elements, so the availability of a particular extension element can be inferred based on the version of Junos that is in use, making this function of limited value; however, using **element-available**() to check on the availability of a EXSLT element would be one way to determine if the Junos version in use is less than 9.4 and therefore lacks support for EXSLT.

Note that at the time of this writing, the **element-available**() function does not automatically convert a result tree fragment or node-set variable to a string data type, so if the desired element name is recorded in a variable with a data type other than string then it must be converted manually via the **string**() function before providing its value to the **element-available**() function.

Example

This example shows multiple invocations of the **element-available**() function, demonstrating the values it returns.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns func extension = "http://exslt.org/functions";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

         /* The following elements are instruction elements, which return true */
        <output> "xsl:sort " _ element-available( "xsl:sort" );
        <output> "exsl:document " _ element-available( "exsl:document");
        <output> "func:result " _ element-available( "func:result");

        /* func:function is a top-level element, so the function returns false */
        <output> "func:function " _ element-available( "func:function");

        /* An example of an element that doesn't exist, so the function returns false */
        <output> "xsl:fake-element " _ element-available( "xsl:fake-element");
    }
}
```

Output

```
xsl:sort true
exsl:document true
func:result true
func:function false
xsl:fake-element false
```

exsl:node-set()

Source: EXSLT
Namespace: http://exslt.org/common
Common Prefix: exsl
Minimum Version: Junos 9.4

Syntax

node-set **exsl:node-set**(object)

Description

The **exsl:node-set**() function is used to convert a result tree fragment, string, or other data type into a node-set. The behavior differs depending on the data type of the argument:

- Result tree fragment - A new XML document is created consisting of the nodes expressed by the result tree fragment. The returned node-set contains the root node of the new XML document.
- Node-set – The node-set argument is returned as the result.
- String – A new XML document is created that consists of a single text node with the string's value, and the returned node-set contains the text node (not the root node).
- Boolean – Value is converted and processed like a string.
- Number – Value is converted and processed like a string.

One common stumbling block when using the exsl:node-set() function is remembering that a converted result tree fragment results in a node-set that contains the root node of the new XML document rather than all of the XML children. For example, the following result tree fragment:

```
var $rtf = {
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/1";
}
```

would be converted into a node-set with a single node: the root-node of the XML document, of which both of the <interface> nodes would be children; however, if a script writer wishes to have the node-set contain all the top-level child nodes, rather than the root node, then the following code can be used:

```
var $ns = exsl:node-set( $rtf )/*;
```

The above example first converts the result tree fragment variable into a node-set, the returned root node is then subjected to a location-path to retrieve its children, and it is these child nodes that are assigned as a node-set to the $ns variable.

Example

This op script demonstrates how **exsl:node-set**() can be used both to convert a result tree fragment variable into a node-set as well as a string into a node-set.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Create RTF */
        var $rtf = {
            <color> "red";
            <color> "blue";
            <color> "green";
        }

        /* Convert to node-set - node in node-set will be root-node of new document */
        var $ns = exsl:node-set( $rtf );

        /* Display color values */
        for-each( $ns/color ) {
            <output> .;
        }

        /*
         * jcs:break-lines() looks at the text node child of the node-set's node, but
         * exsl:node-set( string ) creates a node-set consisting of the text node itself,
         * rather than the parent of the text node, so take a step back to the parent and
         * have jcs:break-lines() process the root node parent of the converted text node
         */
        var $lines = jcs:break-lines( exsl:node-set( "Line 1\nLine 2" )/.. );

        /* Display lines */
        for-each( $lines ) {
            <output> .;
        }
    }
}
```

Output

```
red
blue
green
Line 1
Line 2
```

exsl:object-type()

Source: EXSLT
Namespace: http://exslt.org/common
Common Prefix: exsl
Minimum Version: Junos 9.4

Syntax

string *[type]* **exsl:object-type**(object)

Description

The **exsl:object-type**() function reports the data type of the argument value provided. The returned string is one of the following six values:

- RTF (Result tree fragment)
- node-set
- string
- boolean
- number
- external (The only external object is the Saxon stored expression)

Example

This op script shows the six possible return values of the **exsl:object-type**() function, including the rarely seen stored expression (external object) data type.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl = "http://exslt.org/common";
ns saxon = "http://icl.com/saxon";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* result tree fragment */
        var $rtf = {
            <one>;
            <two>;
            <three>;
        }
        <output> exsl:object-type( $rtf );

        /* Node-set */
```

```
            var $ns = exsl:node-set( $rtf );
            <output> exsl:object-type( $ns );

            /* Boolean */
            var $boolean = ( 5 == 4 );
            <output> exsl:object-type( $boolean );

            /* Number */
            var $number = 100;
            <output> exsl:object-type( $number );

            /* String */
            var $string = "I'm a string";
            <output> exsl:object-type( $string );

            /* External - Saxon stored expression */
            var $expression = saxon:expression("jcs:output( name() )" );
            <output> exsl:object-type( $expression );
        }
    }
```

Output

```
RTF
node-set
boolean
number
string
external
```

false()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **false**()

Description

The **false**() function returns the boolean value of false.

Example

This op script example demonstrates how the **false**() function can be used to assign a boolean value of false to a template parameter.

Code

```
version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        call display-boolean( $boolean = false() );
    }
}

template display-boolean( $boolean ) {
    if( $boolean ) {
        <output> "Boolean value is true";
    }
    if( not( $boolean ) ) {
        <output> "Boolean value is false";
    }
}
```

Output

```
Boolean value is false
```

floor()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **floor**(number)

Description

The **floor**() function is one of the available SLAX rounding functions. It rounds the number argument down to return the largest integer that is not greater than the argument.

Example

This example shows the result of the **floor**() function with many different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> floor( 1 );
        <output> floor( 0.9 );
        <output> floor( 5.1 );
        <output> floor( 5.5 );
        <output> floor( 5.999999999 );
        <output> floor( -3.9 );
        <output> floor( -0.9 );
        <output> floor( -3.1 );
    }
}
```

Output

```
1
0
5
5
5
-4
-1
-4
```

format-number()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string *[number]* **format-number**(number, string *[pattern]*)

string *[number]* **format-number**(string *[number]*, string *[pattern]*)

string *[number]* **format-number**(number, string *[pattern]*, string *[decimal-format]*)

string *[number]* **format-number**(string *[number]*, string *[pattern]*, string *[decimal-format]*)

Description

The **format-number**() function converts a number into a string based on the provided pattern and decimal-format. The first argument is the number to convert, the second argument is the pattern string that should control the conversion, and the third argument, if present, is the name of the decimal-format to use. If the third argument is not present then the default decimal-format is used.

The first argument, which is the number to be formatted, can be provided as either a number or a string argument. If the number is provided as a string then it can be in E notation rather than normal decimal

notation: “10.1E+5” or “10.1e+5” instead of 1010000.

The format pattern string, the second argument, dictates how the number should be formatted, using special characters defined by the selected decimal-format. The relevant characters of a decimal-format are:

- decimal-separator – defaults to .
- grouping-separator – defaults to ,
- percent – defaults to %
- per-mille – defaults to ‰ (Unicode #x2030)
- zero-digit – defaults to 0
- digit – defaults to #
- pattern-separator – defaults to ;

For example, to cause the number 500000 to be displayed as 500,000.00 the following pattern string would be used (along with the default decimal-format):

```
format-number( 500000, "###,###.00" )
```

The digit character in the pattern string indicates an optional digit, but the zero-digit character indicates a required digit, so its value is substituted if no corresponding digit exists in the number being formatted. For example, the following provides a result of “001”:

```
format-number(1, "000" )
```

But this provides a result of “1”:

```
format-number(1, "###" )
```

Modifying the zero-digit in the decimal-format effects both the pattern string as well as the formatted number string, because the characters used for all digits are relative to the character used as the zero-digit, meaning that if the zero-digit was set in the decimal-format as “a”, then 1 would be “b”, 2 would be “c”, etc. Due to this behavior, it is rarely useful to modify the zero-digit within the decimal-format.

There is one special case where the digit character is treated as a required digit: “.#” is treated as “.0” (“.##” is treated as “.0#”, etc.). This means that the zero-digit character will be appended as the initial decimal digit if no corresponding digit exists in the number. For example, the following provides a result of “1.0”:

```
format-number(1, ".#")
```

Integers are never truncated; even if they contain more digits than are covered by the pattern string. Decimals, however, are truncated and rounded based on the number of digits/zero-digits following the decimal-separator in the pattern.

Only the decimal-separator, grouping-separator, and pattern-separator characters are allowed between the digit and zero-digit characters in the pattern string. Other characters can only be used as either suffixes or prefixes.

The decimal-separator and grouping-separator characters are used both for parsing the pattern string as well as for formatting the number string. In other words, by default, the decimal-separator character is “.”, so the presence of “.” in the pattern divides the pattern into an integer and decimal portion. If, however, a decimal-format is used where the decimal-separator is specified as “,” then that character, instead of the default “.”, is used to determine the location of the decimal within the pattern string as well. What this means is that you could not have “100,000.00” as the pattern string, but get this as the converted number, “100.000,00”, because the decimal-separator and grouping-separator have to match between the pattern string and the formatted number.

The percent character can appear at either the beginning or end of the pattern string. If the percent character is

present within the pattern then the formatted number will be the original number argument multiplied by one hundred. The per-mille character can also appear at either the beginning or end of the pattern string. Its presence causes the formatted number to be multiplied by one thousand. The percent and per-mille characters cannot both appear in the same pattern string.

Note: At the time of this writing, Junos cannot display the default per-mille "‰" character via the <output> result tree element and outputs a "0" instead. If you wish to use the per-mille character in a formatted number then you can define a custom number-format with an alternate per-mille character.

The pattern separator is used to define an alternate negative number format. When present the pattern preceding the separator defines the positive format, while the pattern following the separator defines the negative format; however, the negative number pattern only controls the prefix and suffix of the formatted number string. The positive number pattern is used for the actual number formatting whether the number is negative or not.

For example, the following code:

```
format-number( -1.2, "00.00;(#.#)" )
```

Will return "(01.20)" because only the "(" prefix and ")" suffix are used from the negative pattern. When formatting the number itself, the "00.00" positive pattern is used even though the number is negative.

If no separate negative number pattern is provided then the default behavior is to prepend the minus-sign, as determined by the decimal-format, to negative numbers.

Example

This op script demonstrates the effect of various pattern strings on different number arguments.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Alternate decimal format */
<xsl:decimal-format name="alternate" grouping-separator="." decimal-separator=",">;

match / {
    <op-script-results> {

        var $num1 = 12345;
        var $num2 = 0.15;
        var $num3 = -55.5;
        var $num4 = 123456.789;

        /* Integers are not truncated */
        <output> format-number( $num1, "#" );

        /* Decimals are rounded and truncated */
        <output> format-number( $num2, "0.#" );

        /* Default negative number pattern */
        <output> format-number( $num3, "#.#" );
```

```
            /* Alternate pattern */
            <output> format-number( $num3, "#.#;(#)" );

            /* Grouping separator */
            <output> format-number( $num4, "###,###.###" );

            /* Alternate grouping separator */
            <output> format-number( $num4, "###.###,###", "alternate" );

            /* Percentage */
            <output> format-number( $num2, "#%" );
        }
    }
```

Output

```
12345
0.2
-55.5
(55.5)
123,456.789
123.456,789
15%
```

function-available()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **function-available**(string [name])

Description

The **function-available**() function returns true if the string argument matches the qualified name of a function within the function library. Qualified names are the namespace prefix and local name, separated by a colon (except in the case of XSLT and XPath functions, which have no namespace). Example: exsl:object-type, where exsl is the namespace prefix and object-type is the local name. The namespace specified in the string argument provided to **function-available**() must be defined within the script.

Custom functions can be added by using the <func:function> element, so **function-available**() could be useful in verifying that a custom function is available for the script to use.

Note that at the time of this writing, the **function-available**() function does not automatically convert a result tree fragment or node-set variable to a string data type, so if the desired function name is recorded in a variable with a data type other than string then it must be converted manually via the **string**() function before providing its value to the **function-available**() function.

Example

This example shows multiple invocations of the **function-available**() function, demonstrating the values it returns.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns func extension = "http://exslt.org/functions";
ns exsl = "http://exslt.org/common";
ns exam = "http://xml.juniper.net/example";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Standard XPath and XSLT functions */
        <output> "string() " _ function-available( "string" );
        <output> "generate-id() " _ function-available( "generate-id" );

        /* EXSLT functions */
        <output> "exsl:object-type() " _ function-available( "exsl:object-type" );
        <output> "exsl:node-set() " _ function-available( "exsl:node-set" );

        /* Custom functions - one exists, the other does not */
        <output> "exam:present() " _ function-available( "exam:present" );
        <output> "exam:not-present() " _ function-available( "exam:not-present" );

    }
}

/* Define the custom function */
<func:function name="exam:present"> {
    expr jcs:output( "Just a placeholder..." );
}
```

Output

```
string() true
generate-id() true
exsl:object-type() true
exsl:node-set() true
exam:present() true
exam:not-present() false
```

generate-id()

Source: XSLT
Namespace: None
Common Prefix: None

Minimum Version: Junos 8.2

Syntax

string *[ID]* **generate-id**()

string *[ID]* **generate-id**(node-set *[node]*)

Description

The **generate-id**() function returns an identifier string that uniquely identifies a node within a XML document. The identifier returned for a specific node is consistent for the current script execution but may differ each time a script is run even if the XML document is identical. For example, if a commit script used the **generate-id**() function multiple times within the script to retrieve the identifier of the <system> configuration node, then the returned identifier would be the same each time the function was called during the life of the commit script, but the next time a commit was issued the identifier that was returned could be different from the prior time a commit was performed. Because of this, identifiers should not be saved externally and referenced between different script executions.

Junos scripts routinely work with multiple XML documents at the same time. The source tree is a XML document, any node-set returned by a function is a XML document, and any node-set converted from a result-tree fragment through the := operator is a XML document. A node's identifier is only applicable for its own XML document, even if the data expressed by the node is identical in multiple documents. For example, if a commit script requests the configuration through **jcs:invoke**(), then it has two copies of the configuration: one from **jcs:invoke**() and one from the source tree, stored in two separate XML documents. So each copy of the configuration would have separate identifiers that could not be used to refer to nodes in the other copy.

Location paths, however, do not create a new XML document but instead return a set of nodes from within an existing document. This is where the **generate-id**() function can be useful, because a node stored in one node-set can be matched with the same node within a different node-set by comparing the identifiers returned by **generate-id**().

If the **generate-id**() function is called with no argument then it returns the identifier of the context-node. If there is more than one node in the node-set provided as an argument to **generate-id**() then the identifier of the first node in document order is returned. If the node-set argument is empty then a blank string is returned.

This is no relationship between the identifiers created by **generate-id**() and any unique identifiers specified in an external XML document, as retrieved by the **id**() function. No attempt is made to prevent identifier collisions between the two methods, and as such, they should not be used together.

Example

This op script example requests an input string and then displays the unique characters that make up the string. The **generate-id**() function is used to compare nodes between two different node-sets (but the same XML document).

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";
```

```
import "../import/junos.xsl";

match / {

    /* Retrieve input from console */
    var $input = jcs:get-input("Enter string: ");

    /* split into characters */
    var $characters = str:tokenize( $input, "" );

    /* Assemble all the unique characters */
    var $unique-characters := {
        for-each( $characters ) {
            /* Get current ID and value */
            var $current-id = generate-id();
            var $character = .;

            /* Get list of matching characters */
            var $matching-list = $characters[. == $character];

            /* Is the current ID the first ID? */
            var $first-id = generate-id( $matching-list );
            if( $current-id == $first-id ) {
                <character> .;
            }
        }
    }

    /* Display them */
    expr jcs:output("Characters: ");
    for-each( $unique-characters/character ) {
        /* If space then surround in brackets */
        if( . == " " ) {
            expr jcs:output( "[", ., "]" );
        }
        else {
            expr jcs:output( . );
        }
    }
}
```

Output

```
Enter string: What is your favorite color?
Characters:
W
h
a
t
[ ]
i
s
y
o
u
r
f
v
e
c
l
```

?

id()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

node-set **id**(node-set *[ID]*)

node-set **id**(string *[ID]*)

Description

The **id**() function retrieves a node based on its unique ID. This ID is the value of the node's attribute that has been assigned to type ID within the XML document's DTD. The desired ID can be provided as either a string or a node-set. When provided as a string it can consist of a single ID value or multiple ID values separated by spaces. The returned node-set will contain a single node for each ID if a matching node is found in the document. When the argument is provided as a node-set then the returned node-set contains all the nodes with matching IDs for each node's string value within the node-set.

The Junos XML API does not assign unique IDs in the way used by **id**(), so this function is not typically used within Junos scripts. It could, however, be used to find referenced nodes within an XML document that has been read by the **document**() function, so long as the XML document contains the necessary attribute type ID assignments within its DTD. When doing so, be aware that the **id**() function searches within the XML document of the context node, so the context node must be set to a node within the retrieved XML document, such as within a for-each loop, in order for the **id**() function to find the matching node.

Example

This op script example demonstrates how the **id**() function can be used in combination with an external XML document to retrieve nodes based on their unique IDs.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $document = document( "/var/tmp/test.xml" );

    for-each( $document/account-list/user ) {
        expr jcs:output( "User ", name, " is assigned to class ", id( @class-id )/name );
```

```
    }
}
```

test.xml

```
<?xml version="1.0"?>
<!DOCTYPE account-list [
<!ELEMENT account-list (user+, class+)>

<!ELEMENT class (name, permissions)>
<!ATTLIST class class-id ID #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT permissions (#PCDATA)>

<!ELEMENT user (name)>
<!ATTLIST user class-id IDREF #REQUIRED>
]>

<account-list>
    <class class-id="1">
        <name>super-user</name>
        <permissions>all</permissions>
    </class>
    <class class-id="2">
        <name>read-only</name>
        <permissions>view</permissions>
    </class>
    <user class-id="1">
        <name>admin</name>
    </user>
    <user class-id="2">
        <name>test</name>
    </user>
</account-list>
```

Output

```
User admin is assigned to class super-user
User test is assigned to class read-only
```

jcs:break-lines()

aka jcs:break_lines()
Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set *[lines]* **jcs:break-lines**(object)

node-set *[lines]* **jcs:break-lines**(object, object*)

Description

The **jcs:break-lines**() function breaks text content into multiple-lines, with the newline (\n) character used as the delimiter. A node-set is returned, which contains each line of the original text content as a separate node. The returned nodes are ordered according to the placement of the lines in the original text content, so if the results are stored in a variable called $lines then the fifth line could be retrieved with this location path: $lines[5]. Typically only a single argument is provided, but multiple arguments are supported. All arguments must be either node-sets or result tree fragments.

If the argument is a node-set then its nodes must be either element or root nodes that have text node children, because it is the text node children that are processed, and only element or root nodes have text node children. Descendent element nodes are not processed, so it is important to select the exact node that has the text content.

Multiple nodes can be included in the node-set argument. In this situation the text content of all of the nodes is split into lines and returned within the node-set. If more than one argument is provided then the split lines from each argument are also combined and returned as a single node-set.

The node names in the returned node-set are copied from their source, so if the <file-contents> element node is being split, then all of the lines will appear in the returned node-set with a name of <file-contents> as well.

If the argument is a result tree fragment then any text content at the root is split into lines. For example, the following text strings “one\n”, “two\n”, and “three\n” would not be processed because they are assigned to elements, rather than appearing at the root level:

```
var $example-rtf = {
    <node> "one\n";
    <node> "two\n";
    <node> "three\n";
}
```

However, the following strings would be processed as desired because they are at the root:

```
var $example-rtf = {
    expr "one\n";
    expr "two\n";
    expr "three\n";
}
```

When a result tree fragment is split, the node names within the returned node-set are blank.

Example

This op script demonstrates a common use for **jcs:break-lines**(): splitting a file’s content into multiple lines. It then allows the user to select a specific line to display.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";
```

```
match / {
    <op-script-results> {

        /* Read the file */
        var $rpc = {
            <file-get> {
                <filename> "/var/log/syslog";
                /* Leading and trailing newlines aren't added when raw encoding is used */
                <encoding> "raw";
            }
        }
        var $file = jcs:invoke( $rpc );

        /* Split into lines */
        var $lines = jcs:break-lines( $file/file-contents );

        /* Display line count */
        expr jcs:output( "Line count: ", count( $lines ) );

        /* Select a line */
        var $input = jcs:get-input( "Select line to display: " );

        /* Display it if it is valid */
        if( $input <= count( $lines ) && $input > 0 ) {
            /* $input must be cast to a number, otherwise it will be treated as a string */
            expr jcs:output(  $lines[ number( $input ) ] );
        }
        else {
            expr jcs:output( "Invalid line number" );
        }
    }
}
```

/var/log/syslog

```
    Nov 10 21:19:31  srx210 /kernel: %KERN-1-GENCFG: op 2 (Gencfg Blob) failed; err 7 (Doesn't
Exist)
    Nov 10 21:19:34  srx210 utmd[1178]: %DAEMON-5-LIBJSNMP_SA_IPC_REG_ROWS: ns_subagent_register_
mibs: registering 1 rows
    Nov 10 21:19:34  srx210 mcsn[1181]: %DAEMON-3-TASK_SCHED_SLIP: 8 sec scheduler slip, user: 0
sec 121503 usec, system: 0 sec, 107208 usec
    Nov 10 21:19:34  srx210 mcsn[1181]: %DAEMON-6-TASK_TASK_REINIT: Reinitializing
    Nov 10 21:19:40  srx210 rtlogd[1177]: %DAEMON-5-LIBJSNMP_SA_IPC_REG_ROWS: ns_subagent_
register_mibs: registering 1 rows
    Nov 10 21:19:40  srx210 idpd[1176]: %DAEMON-5-LIBJSNMP_SA_IPC_REG_ROWS: ns_subagent_register_
mibs: registering 1 rows
    Nov 10 21:19:43  srx210 idpinfo: %USER-7: pic_info_list_delete: PIC_INFO debug> pic_info_head
0x7aa070
    Nov 10 21:19:43  srx210 idpd[1176]: %DAEMON-5-IDP_COMMIT_COMPLETED: IDP policy commit is
complete.
    Nov 10 21:19:43  srx210 idpd[1176]: %DAEMON-6-LICENSE_CONN_TO_LI_CHECK_SUCCESS: Connected to
license-check
    Nov 10 21:19:47  srx210 srx210 idp_ctrl_handle_connect: %PFE-6: Connected to IDPD
    Nov 10 21:19:47  srx210 srx210 idp_ctrl_handle_socket_event: %PFE-7: Received event 8
    Nov 10 21:19:47  srx210 pkid[1175]: %DAEMON-5-LIBJSNMP_SA_IPC_REG_ROWS: ns_subagent_register_
mibs: registering 2 rows
    Nov 10 21:19:55  srx210 mgd[866]: %INTERACT-5-UI_DBASE_LOGOUT_EVENT: User 'root' exiting
configuration mode
    Nov 10 21:23:30  srx210 mgd[1201]: %INTERACT-6-UI_AUTH_EVENT: Authenticated user 'jnpr' at
permission level 'j-super-user-local'
    Nov 10 21:23:38  srx210 mgd[1201]: %INTERACT-6-UI_CMDLINE_READ_LINE: User 'jnpr', command
'file list /var/tmp '
    Nov 10 21:23:38  srx210 mgd[1201]: %INTERACT-6-UI_CHILD_START: Starting child '/bin/sh'
```

```
    Nov 10 21:23:38  srx210 mgd[1201]: %INTERACT-6-UI_CHILD_STATUS: Cleanup child '/bin/sh', PID
1202, status 0
    Nov 10 21:23:46  srx210 mgd[1201]: %INTERACT-6-UI_CMDLINE_READ_LINE: User 'jnpr', command
'file list /cf/var/tmp '
    Nov 10 21:23:46  srx210 mgd[1201]: %INTERACT-6-UI_CHILD_START: Starting child '/bin/sh'
    Nov 10 21:23:46  srx210 mgd[1201]: %INTERACT-6-UI_CHILD_STATUS: Cleanup child '/bin/sh', PID
1204, status 0
```

Output

```
    Line count: 20
    Select line to display: 2
    Nov 10 21:19:34  srx210 utmd[1178]: %DAEMON-5-LIBJSNMP_SA_IPC_REG_ROWS: ns_subagent_register_
mibs: registering 1 rows
```

jcs:close()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.3

Syntax

node-set *[empty]* **jcs:close**(node-set *[connection]*)

Description

The **jcs:close**() function closes the management session indicated by the connection handle argument. While a node-set is returned, it is always empty whether the close operation was successful or not. If a session indicated by the connection handle does not exist then an error is displayed:

```
    error: Session for server "example" does not exist
```

The jcs:close() function is always paired with the jcs:open() function, which opens the session. The typical process is to open a connection with jcs:open() and assign the returned connection handle to a variable, then provide that connection variable to one or more jcs:execute() function calls in order to run multiple Junos API requests within a single management session. Finally, after all processing is done, the jcs:close() function is called, including the connection handle as its argument, in order to close the session.

Example

This op script demonstrates how to use the **jcs:close**() function.

Code

```
    version 1.0;

    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Open a session */
        var $connection = jcs:open();

        /* Example of a command that could be run with the connection handle */
        var $operation = jcs:execute( $connection, "get-software-information" );

        /* Copy contents to the screen */
        copy-of $operation;

        /* Close the session */
        expr jcs:close( $connection );
    }
}
```

Output

```
Hostname: srx210
Model: srx210h
JUNOS Software Release [10.3R1.9]
```

jcs:dampen()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.4

Syntax

boolean **jcs:dampen**(string *[tag]*, number *[rate]*, number *[minutes]*)

Description

The **jcs:dampen**() function is normally used to limit the number of times a particular code-block is executed. The first argument is the tag string, the second argument is the rate, and the third argument is the minute interval. The function returns true if it has not been called successfully for the tag string more than the times specified by the rate value within the given interval value. If the rate has been exceeded then it returns false.

A typical use would be like this:

```
if( jcs:dampen( "example-tag", 2, 1 ) ) {
    /* Only execute this code twice per minute */
    ...
}
```

In the example above, the code within the if code-block is only executed if the script is called a maximum of two times within a minute interval. Otherwise jcs:dampen() returns false and causes the code-block to not be run.

The tag string scope is defined by the script's name and type, but not by a specific invocation of the script; rather, it is designed to persist between multiple calls of the same script, which makes it particularly useful for event scripts where you only want to execute the event script code a certain number of times within a given time period, rather than every time the script is called.

So, an event script of "test.slax" has a separate tag string space from an event script called "example.slax", or from an op script called "test.slax", but each time **jcs:dampen**() is called by the event script "test.slax" it uses the same tag string space.

Note: Op scripts that are executed by event policies are considered to be of type op script as far as the tag string scope is concerned.

Unsuccessful calls to **jcs:dampen**() (when it returns false because the rate has been exceeded) have no effect on later calls. In other words, the function tests for a maximum number of successful calls to **jcs:dampen**() within a time period, as opposed to the total count of both successful and unsuccessful calls.

Example

This op script is an example of how a login script could take advantage of **jcs:dampen**() to provide a message if a user has logged in too many times within a particular time interval.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /*
         * If a specific user has logged in more than three times in five
         * minutes then display a warning message to them and the syslog.
         */
        if( not( jcs:dampen( $user, 3, 5 ) ) ) {
            <output> "Warning: You have exceeded your login quota";
            expr jcs:syslog( "daemon.warning", $user, " has exceeded their login quota" );
        }
    }
}
```

Output

```
jnpr@srx210> ssh 10.0.0.10
jnpr@10.0.0.10's password:
--- JUNOS 10.3R1.9 built 2010-08-13 13:07:06 UTC

jnpr@srx210> exit
Connection to 10.0.0.10 closed.

jnpr@srx210> ssh 10.0.0.10
jnpr@10.0.0.10's password:
--- JUNOS 10.3R1.9 built 2010-08-13 13:07:06 UTC
```

```
    jnpr@srx210> exit
    Connection to 10.0.0.10 closed.

    jnpr@srx210> ssh 10.0.0.10
    jnpr@10.0.0.10's password:
    --- JUNOS 10.3R1.9 built 2010-08-13 13:07:06 UTC

    jnpr@srx210> exit
    Connection to 10.0.0.10 closed.

    jnpr@srx210> ssh 10.0.0.10
    jnpr@10.0.0.10's password:
    --- JUNOS 10.3R1.9 built 2010-08-13 13:07:06 UTC
    Warning: You have exceeded your login quota

    jnpr@srx210> show log messages | last | match cscript
    Nov 13 00:23:04  srx210 cscript: jnpr has exceeded their login quota
    Nov 13 00:23:34  srx210 mgd[1515]: UI_CMDLINE_READ_LINE: User 'jnpr', command 'show log
messages | last | match cscript '
```

jcs:empty()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

boolean **jcs:empty**(object*)

Description

The **jcs:empty**() function is used to determine if an object is empty. It is typically called with a single argument, but any number of arguments are supported. It returns a boolean value of true if all of the arguments are empty; otherwise, if any are not empty, then it returns a value of false.

At the time of this writing, only two data types are supported: node-sets and strings. A node-set is considered empty if it has no nodes. A string is considered empty if it is blank. Number and boolean data types are always considered empty, and result tree fragments are always considered to not be empty. The latter behavior can be confusing because result tree fragment variables are often treated by script writers as if they were strings, but any time a variables value is assigned through the use of curly brackets, such as when assigning the results of a template to a variable, it is actually a result tree fragment, not a string, even if it only contains text content, which means that **jcs:empty**() currently considers it to never be empty, even if it has no elements or text content.

Example

This commit script shows how the **jcs:empty**() function can be used to indicate if a configuration statement is missing. In this case, the script is checking for the presence of a system host-name.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {

    /* Declare an error if the host-name is missing */
    if( jcs:empty( system/host-name ) ) {
        <xnm:error> {
            <message> "No host-name is configured";
        }
    }
}
```

Output

```
[edit]
jnpr@srx210# show system host-name

[edit]
jnpr@srx210# commit
error: No host-name is configured
error: 1 error reported by commit scripts
error: commit script failure
```

jcs:execute()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.3

Syntax

node-set *[result]* **jcs:execute**(object *[connection]*, object *[RPC]*)

Description

The **jcs:execute**() function is used to execute a RPC from the Junos API within a specific management session, in contrast to the **jcs:invoke**() function, which also performs API calls but uses a unique session per command. This difference makes **jcs:execute**() the appropriate function to use when making configuration changes, because a safe configuration change requires multiple commands to be executed within the same session: locking the database, loading the configuration, committing the configuration, and then unlocking the database. In addition, if the management session is opened to a remote Junos device, then **jcs:execute**() can be used to execute RPCs remotely, but **jcs:invoke**() can only interact with the local Junos device.

The first argument to **jcs:execute**() is the connection handle, which must have been opened via **jcs:open**() prior

to calling **jcs:execute**(). Typically this argument is a node-set variable that was returned when the connection was opened by the **jcs:open**() function; however, a string is also accepted so long as it matches the cookie value of a current connection handle. (The cookie of a local connection is a blank string. The cookie of a remote connection is the hostname/address used in the call to **jcs:open**()).

If the connection handle is not already opened then the script fails with an error message:

```
error: Session for server "" does not exist
error: xmlXPathCompiledEval: evaluation failed
```

The second argument to **jcs:execute**() is the RPC that should be invoked, provided as either a string, a result tree fragment, or a node-set. If provided as a string then it must match the name of the RPC, and no attributes or child elements can be provided. For example, this would execute the <get-software-information> RPC:

```
var $result = jcs:execute( $connection, "get-software-information" );
```

If the second argument is either a node-set or a result tree fragment then Junos will attempt to execute the RPC described by the first child element node. (Multiple RPCs cannot be executed through a single call to **jcs:execute**(). When using a node-set argument remember to provide the parent node of the RPC element node, which could be the root node, rather than the RPC element node itself). The advantage of using one of these two data-types is that attributes, child elements, and text content can be provided for the RPC, which is not possible with a string argument. Here is an example where the RPC is specified as a result tree fragment so that the database and child element filters can be specified:

```
var $config-rpc = {
    <get-configuration database="committed"> {
        <configuration> {
            <system>;
        }
    }
}
var $system-configuration = jcs:execute( $connection, $config-rpc );
```

The result is returned as a node-set, which contains the child node of the <rpc-reply> generated in response to the RPC request. The node-set can be empty if the Junos API does not return results for the requested command. If the RPC is invalid in some way then an error message is usually indicated by the presence of a <xnm:error> node within the XML document referenced by the returned node-set.

Example

This op script example demonstrates how to use **jcs:execute**() to request that a portion of the configuration be retrieved.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Open a local connection */
        var $connection = jcs:open();
```

```
        /* Retrieve the configured AS Number */
        var $configuration-rpc = {
            <get-configuration database="committed" inherit="inherit"> {
                <configuration> {
                    <routing-options> {
                        <autonomous-system>;
                    }
                }
            }
        }
        var $results = jcs:execute( $connection, $configuration-rpc );
        var $asn = $results/routing-options/autonomous-system/as-number;

        /* Close the connection */
        expr jcs:close( $connection );

        /* Display the results */
        if( $asn ) {
            <output> "AS Number is " _ $asn;
        }
        else {
            <output> "No AS is configured";
        }
    }
}
```

routing-options Configuration

```
routing-options {
    autonomous-system 65535;
}
```

Output

```
AS Number is 65535
```

jcs:first-of()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

object **jcs:first-of**(object*)

Description

The **jcs:first-of**() function returns the first non-empty argument from its argument list. It is particularly useful when dealing with configuration statements that can be overridden by statements at a more preferred hierarchy level. An example of this is BGP configuration where the neighbor configuration overrides the group configuration, which overrides the general BGP configuration, so using the **jcs:first-of**() function provides a simple way to select the configuration statement that actually gets applied. The code example for this function demon-

strates how this can be done.

Any number of arguments can be provided, but only the node-set and string data-type are supported. A node-set is considered to be not empty if it contains one or more nodes. A string is considered to be not empty if it is not blank. Result tree fragments are not supported as of the time of this writing, and they are always considered to be non-empty, even if they have no contents.

The returned data-type depends on the selected non-empty argument, but it should be either a node-set or a string as those are the only supported data types at the time of this writing. If there are no non-empty arguments in the argument list then an empty node-set is returned.

Example

This op script uses the **jcs:first-of**() function to select the correct import policies for a BGP peer and then displays them.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve desired peer */
        var $peer = jcs:get-input( "Enter BGP peer address: ");

        /* Get the configuration */
        var $rpc = <get-configuration database="committed" inherit="inherit">;
        var $configuration = jcs:invoke( $rpc );

        /* Retrieve the peer config */
        var $peer-config = $configuration/protocols/bgp//neighbor[name == $peer];

        /* If peer is not found then give an error */
        if( jcs:empty( $peer-config ) ) {
            <output> "Error: BGP peer " _ $peer _ " was not found";
        }
        /* Otherwise, display the import policies */
        else {
            /* Find the correct import policy configuration for the peer */
            var $import = jcs:first-of( $peer-config/import, $peer-config/../import,
                                        $peer-config/../../import, "*None*" );

            /* Handle node-sets differently than strings */
            var $policy-string = {
                /* If a node-set, then add all the values */
                if( exsl:object-type( $import ) == "node-set" ) {
                    for-each( $import ) {
                        expr . _ " ";
                    }
                }
```

```
                /* If a string then just add */
                else {
                    expr $import;
                }
            }

            <output> "Import policies: " _ $policy-string;
        }
    }
}
```

protocols configuration

```
protocols {
    bgp {
        import BLOCK-LOCAL;
        group AS65500 {
            import [ BLOCK-LOCAL BLOCK-PRIVATE ];
            peer-as 65500;
            neighbor 10.0.0.101;
            neighbor 10.0.0.102 {
                import [ BLOCK-LOCAL BLOCK-PRIVATE REMOVE-MED ];
            }
        }
        group AS65490 {
            peer-as 65490;
            neighbor 10.0.0.103;
        }
    }
}
```

Output

```
Enter BGP peer address: 10.0.0.102
Import policies: BLOCK-LOCAL BLOCK-PRIVATE REMOVE-MED
```

jcs:get-input()

aka jcs:input()
Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.6
(jcs:input(): 9.4)

Syntax

string **jcs:get-input**(string *[prompt]*)

Description

The **jcs:get-input**() function displays a prompt string and then allows the user to input a string, terminated by the enter/return key. The single argument provided to the function is the prompt string displayed to the user. The prompt has a maximum length of 1024 characters and is truncated if the provided string is too long.

The user's input is displayed as it is typed, so **jcs:get-input**() should not be used to retrieve sensitive information such as passwords. Instead, sensitive input should be retrieved through the **jcs:get-secret**() function, which does not display the user input.

jcs:get-input() is intended primarily for op scripts, and is always unsupported in event scripts; however, it can be used within commit scripts so long as the commit was invoked by a user through the Junos CLI, and the script is running on the local routing-engine, but it is unsupported in all other scenarios: automatic commits such as at boot-up, commits performed via the Junos API, and commits on non-local routing-engines. Typically, when **jcs:get-input**() is run in an unsupported scenario, no prompt string is displayed and an empty node-set, equivalent to a zero-length string, is returned; but if this function is used by an op script that is invoked by an event policy, and thus is acting as an event script, then the script terminates with an error.

Example

This op script shows an example of using the **jcs:get-input**() and **jcs:get-secret**() functions to retrieve the necessary information to log into a remote device.

Code

```
    version 1.0;


    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match / {
        <op-script-results> {

            /*
             * Gather information for remote login. The first two are not secret
             * so use the jcs:get-input() function, which displays the entered text,
             * but the password is retrieved with the jcs:get-secret() function so
             * that its text is not displayed.
             */
            var $remote-host = jcs:get-input( "Enter host-name: " );
            var $user-name = jcs:get-input( "Enter user-name: " );
            var $password = jcs:get-secret( "Enter password: " );

            /* Open the remote connection */
            var $connection = jcs:open( $remote-host, $user-name, $password );

            if( $connection ) {
                var $result = jcs:execute( $connection, "get-software-information" );

                /* Grab the version - This path should work on both single and multi-re systems
*/
                var $version =
                    $result/..//software-information[1]/package-information[name == "junos"];
                expr jcs:output( "Junos version on remote host: ", $version/comment );
            }
            else {
                expr jcs:output( "Unable to open a connection." );
            }
        }
    }
```

Output

```
Enter host-name: 10.0.0.10
Enter user-name: jnpr
Enter password:
Junos version on remote host: JUNOS Software Release [10.4R1.9]
```

jcs:get-secret()

aka jcs:getsecret()
Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.6

Syntax

string **jcs:get-secret**(string *[prompt]*)

Description

The **jcs:get-secret**() function displays a prompt string and then allows the user to input a string, terminated by the enter/return key. The single argument provided to the function is the prompt string displayed to the user. The prompt has a maximum length of 1024 characters and is truncated if the provided string is too long.

The user's input is not displayed as it is typed, making **jcs:get-secret**() ideal for entering passwords and other information; however, the **jcs:get-input**() function can be used, instead, if the input should be displayed as it is typed.

jcs:get-secret() is intended primarily for op scripts, and is always unsupported in event scripts; however, it can be used within commit scripts so long as the commit was invoked by a user through the Junos CLI, and the script is running on the local routing-engine, but it is unsupported in all other scenarios: automatic commits such as at boot-up, commits performed via the Junos API, and commits on non-local routing-engines. Typically, when **jcs:get-secret**() is run in an unsupported scenario, no prompt string is displayed and an empty node-set, equivalent to a zero-length string, is returned but if this function is used by an op script that is invoked by an event policy, and thus is acting as an event script, then the script terminates with an error.

Example

This op script shows an example of using the **jcs:get-input**() and **jcs:get-secret**() functions to retrieve the necessary information to log into a remote device.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
    import "../import/junos.xsl";

    match / {
        <op-script-results> {

            /*
             * Gather information for remote login. The first two are not secret
             * so use the jcs:get-input() function, which displays the entered text,
             * but the password is retrieved with the jcs:get-secret() function so
             * that its text is not displayed.
             */
            var $remote-host = jcs:get-input( "Enter host-name: " );
            var $user-name = jcs:get-input( "Enter user-name: " );
            var $password = jcs:get-secret( "Enter password: " );

            /* Open the remote connection */
            var $connection = jcs:open( $remote-host, $user-name, $password );

            if( $connection ) {
                var $result = jcs:execute( $connection, "get-software-information" );

                /* Grab the version - This path should work on both single and multi-re systems
*/
                var $version =
                    $result/..//software-information[1]/package-information[name == "junos"];
                expr jcs:output( "Junos version on remote host: ", $version/comment );
            }
            else {
                expr jcs:output( "Unable to open a connection." );
            }
        }
    }
```

Output

```
    Enter host-name: 10.0.0.10
    Enter user-name: jnpr
    Enter password:
    Junos version on remote host: JUNOS Software Release [10.4R1.9]
```

jcs:hostname()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

string **jcs:hostname**(string)

Description

The **jcs:hostname**() function returns the fully qualified domain name for the hostname, IPv4 address, or IPv6

address that is provided in the string argument. The fully qualified domain name is retrieved through static host mapping within the Junos configuration or by querying the configured name servers.

A blank string is returned if no fully qualified domain name can be found for the hostname or address.

Example

This op script displays the fully qualified domain name that corresponds to the provided command-line argument.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* CLI argument */
var $argument = {
    <argument> {
        <name> "value";
        <description> "Address or hostname to check";
    }
}
param $value;

match / {
    <op-script-results> {
        var $result = jcs:hostname( $value );

        <output> "jcs:hostname() result is " _ $result;
    }
}
```

Output

```
jnpr@srx210> op jcs_hostname value srx210
jcs:hostname() result is srx210.example.com

jnpr@srx210> op jcs_hostname value 10.0.0.2
jcs:hostname() result is srx210-2.example.com

jnpr@srx210> op jcs_hostname value 2001:db8::2650
jcs:hostname() result is srx650.example.com
Junos version on remote host: JUNOS Software Release [10.4R1.9]
```

jcs:invoke()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs

Minimum Version: Junos 8.2

Syntax

node-set *[result]* **jcs:invoke**(string *[RPC]*)

node-set *[result]* **jcs:invoke**(result-tree-fragment *[RPC]*)

node-set *[result]* **jcs:invoke**(node-set *[RPC]*)

Description

The **jcs:invoke**() function is used to execute a RPC from the Junos API within a unique management session, in contrast to the **jcs:execute**() function, which also performs API calls but executes them all within a single management session. While creating/closing a management session each time a RPC is executed does cause **jcs:invoke**() to take slightly more processing time than **jcs:execute**(), the ability to use the function without manually creating and closing a management session makes it useful for scripts that only execute a few RPCs; however, if there are a large quantify of RPCs being executed then the slight difference in processing time becomes more apparent and **jcs:execute**() should be used instead. Also, commits require multiple RPCs to be executed within the same management session, so they must only be performed by **jcs:execute**(), and **jcs:invoke**() is only capable of executing RPCs on the local Junos device. If a RPC should be executed on a remote Junos device then the **jcs:execute**() function is needed instead.

The single argument to **jcs:invoke**() is the RPC that should be invoked, provided as either a string, a result tree fragment, or a node-set. If provided as a string then it must match the name of the RPC, and no attributes or child elements can be provided. For example, this would execute the <get-software-information> RPC:

```
var $result = jcs:invoke( "get-software-information" );
```

If the argument is either a node-set or a result tree fragment then Junos will attempt to execute the RPC described by the first child element node. (Multiple RPCs cannot be executed through a single call to **jcs:invoke**(). When using a node-set argument remember to provide the parent node of the RPC element node, which could be the root node, rather than the RPC element node itself). The advantage of using one of these two data-types is that attributes, child elements, and text content can be provided for the RPC, which is not possible with a string argument. Here is an example where the RPC is specified as a result tree fragment so that the database and child element filters can be specified:

```
var $config-rpc = {
    <get-configuration database="committed"> {
        <configuration> {
            <system>;
        }
    }
}
var $system-configuration = jcs:invoke( $config-rpc );
```

The result is returned as a node-set, which contains the child node of the <rpc-reply> generated in response to the RPC request. The node-set can be empty if the Junos API does not return results for the requested command. If the RPC is invalid in some way then an error message is usually indicated by the presence of a <xnm:error> node within the XML document referenced by the returned node-set.

Example

This op script example demonstrates how to interact with the Junos API using the **jcs:invoke**() function.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* String example */
        var $results1 = jcs:invoke( "get-software-information" );

        /* RTF example */
        var $command = <command> "show system users";
        var $results2 = jcs:invoke( $command );

        /* Node-set example */
        var $command-set := {
            <get-interface-information> {
                <terse>;
                <interface-name> "ge-0/0/0";
            }
        }
        var $results3 = jcs:invoke( $command-set );

        /* Display results on the terminal */
        <output> "show version:";
        copy-of $results1;

        <output> "show system users:";
        copy-of $results2;

        <output> "show interfaces terse";
        copy-of $results3;
    }
}
```

Output

```
show version:
Hostname: srx210
Model: srx210h
JUNOS Software Release [10.4R1.9]
show system users:
10:53PM  up 1:35, 1 user, load averages: 0.26, 0.10, 0.08
USER     TTY      FROM                              LOGIN@  IDLE WHAT
jnpr     p1       10.0.0.50                        10:33PM     - -cli (cli)
show interfaces terse
Interface               Admin Link Proto    Local                 Remote
ge-0/0/0                up    up
ge-0/0/0.0              up    up   inet     10.0.0.10/24
```

jcs:open()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.3

Syntax

node-set *[connection]* **jcs:open**()

node-set *[connection]* **jcs:open**(string *[host]*)

node-set *[connection]* **jcs:open**(string *[host]* , string *[user]*, string *[password]*)

Description

The **jcs:open**() function opens a local or remote management session within which RPCs can be executed through the **jcs:execute**() function. If the session opens successfully then **jcs:open**() returns a node-set connection handle, which is used by the **jcs:execute**() to identify the management session. Otherwise, if the connection fails then an empty node-set is returned, and an error is displayed to the executing user if the failure occurred within an op script.

Typically, a management session is opened through **jcs:open**() and the returned connection handle assigned to a variable. Then one or more RPCs are run within the session by the **jcs:execute**() function. Finally, when the session is no longer needed, it is closed by the **jcs:close**() function.

Whether the connection is local or remote is determined by the number of arguments provided. If the session is remote then the authentication used is determined by the number of arguments as well.

If no arguments are provided, then a local management session is opened. This can be done within op scripts, event scripts, or commit scripts:

```
var $local-connection = jcs:open();
```

With ssh-agent forwarding enabled and key-based authentication in place, an op script can create a management session on a remote Junos device by providing the remote hostname as the single argument to jcs:open(). The local user name is used to log into the remote host, and authentication is performed through ssh keys:

```
var $remote-connection = jcs:open( "remote-host" );
```

In addition, as of Junos 10.0 (9.5R3, 9.6R3), an op script can use the above syntax without using key-based authentication, because a password prompt is provided directly to the CLI user when key-based authentication is unavailable. This allows the user to provide the necessary password to log into the remote device, after which the script can perform its processing within the remote management session.

The final method to open a remote management session is to provide **jcs:open**() both the username and password along with the hostname. This approach is supported as of Junos 9.6 and can be used by op scripts, event scripts, and commit scripts:

```
var $remote-connection = jcs:open("remote-host", "user", "password");
```

Remote connections are performed with SSH, so the host key of the remote device must be in the known_hosts file of the script's executing user, which could be root in some cases, or it must be configured within the [edit security ssh-known-hosts] Junos configuration hierarchy. For commit and event scripts, the host key must be in place prior to script execution, but as of Junos 10.0 (9.5R3, 9.6R3), a prompt is provided, if

necessary, to the op script's executing user, allowing the user to approve the key and automatically add it to their known_hosts file.

Note: As of the time of this writing, the prompt to accept a new host key is only displayed when **jcs:open**() is called with a single argument.

A change of the remote host's key (i.e. a "man-in-the-middle" attack warning) cannot be reconciled within an op script and must be handled manually before the script is run.

Example

This op script example demonstrates how **jcs:open**() can open a remote management session and execute commands within it.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* CLI argument */
var $arguments = {
    <argument> {
        <name> "remote-host";
        <description> "Address or host to connect to";
    }
}
param $remote-host;

match / {
    <op-script-results> {

        /* Open a local session */
        var $local-connection = jcs:open();

        /* Retrieve the local software version and hostname */
        var $local-results = jcs:execute( $local-connection, "get-software-information" );

        /* Display the local version */
        var $local-version = $local-results/package-information[name == "junos"]/comment;
        var $local-host-name = $local-results/host-name;
        <output> "Local host-name: " _ $local-host-name _ " Version: " _ $local-version;

        /* Close the local session */
        expr jcs:close( $local-connection );

        /* Open a remote session using ssh-agent forwarding */
        var $remote-connection = jcs:open( $remote-host );

        /* Retrieve the remote software version and hostname */
        var $remote-results = jcs:execute( $remote-connection, "get-software-information" );

        /* Display the remote version */
        var $remote-version = $remote-results/package-information[name == "junos"]/comment;
        var $remote-host-name = $remote-results/host-name;
        <output> "Remote host-name: " _ $remote-host-name _ " Version: " _ $remote-version;
```

```
        /* Close the remote session */
        expr jcs:close( $remote-connection );

    }
}
```

Output

```
jnpr@R2D2-M320-RE0> op jcs_open remote-host jawa
Local host-name: R2D2-M320-RE0 Version: JUNOS Base OS boot [10.3R1.9]
Remote host-name: Jawa-RE0 Version: JUNOS Base OS boot [11.2-20101220.0]
```

jcs:output()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set *[empty]* **jcs:output**(string)

node-set *[empty]* **jcs:output**(string, string*)

Description

The **jcs:output**() function displays one or more lines of output text, either to the CLI user (when used in op scripts), or to the output file (when used in event scripts). It can be called with only a single string argument, or multiple arguments can be provided, causing them to be concatenated into one combined string. A newline always terminates the output text, so each call to **jcs:output**() occurs on a separate line.

jcs:output() is not supported in commit scripts. They use the <xnm:warning> and <xnm:error> result tree elements to display text to the CLI user.

The behavior of **jcs:output**() differs from the **<output>** result tree element in that **jcs:output**() displays its text immediately, rather than waiting until the conclusion of the script. This makes it suitable for scripts where user interaction is required, such as when the **jcs:get-input**() function is used, or when status messages should be displayed in the midst of the script processing.

While **jcs:output**() does return a node-set, it is always empty and can be ignored, so **jcs:output**() is normally called with the **expr** statement, rather than assigning its result to a variable:

```
expr jcs:output( "This is an example" );
```

The following escape characters are supported in the output text:

- \n – Newline
- \r – Carriage Return
- \t - Tab

- \\ - Backslash (As of Junos 10.2)
- \" – Double-quote (As of Junos 10.1R2)
- \' – Single-quote

As of Junos 10.2, the maximum length for the output text is 10 KB, and longer strings are truncated to the supported length.

Note: As of the time of this writing, a double-quote and a single-quote cannot both appear in the same string, and leading and terminating whitespace is generally not displayed.

Example

This op script example demonstrates how to display text with **jcs:output**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    expr jcs:output( "Script Details:" );
    expr jcs:output( "Name is: ", $script );
    expr jcs:output( "Run on: ", $hostname );
    expr jcs:output( "Run by: ", $user );
    expr jcs:output( "Date: ", $localtime-iso );
}
```

Output

```
Script Details:
Name is: jcs_output.slax
Run on: srx210
Run by: jnpr
Date: 2011-01-05 19:02:36 UTC
```

jcs:parse-ip()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.0

Syntax

node-set **jcs:parse-ip**(string *[address]*)

Description

The **jcs:parse-ip**() function analyzes an IPv4 or IPv6 address and returns its details within a structured node-set. The string argument that is provided to the function is an IPv4 or IPv6 address. Ideally, the address should contain a prefix-length or network mask (IPv4 only), because if it does not, then the **jcs:parse-ip**() function only returns the host address and address family, and the rest of the details described below are not provided.

Here are examples of acceptable addresses:

- 10.100.100.1/24
- 192.168.1.1/255.255.255.0
- 2001:db8::100:1/64

The return value is a structured node-set, consisting of four or five <parse-ip> element nodes, which are arranged in the following order:

- Host address
- Address family (inet or inet6)
- Prefix length
- Network address
- Network mask (IPv4 only)

Because the results are always provided in the same position, the particular data of interest can be retrieved by using the [#] location path notation. For example, if the results of the **jcs:parse-ip**() function were assigned to a variable named $results then the following location path would return the prefix length value: $results[3].

If the provided address is invalid, then an empty node-set is returned and an error message is generated similar to the following:

```
error: parse-ip error: invalid input at '.1' in ip address '10.0.0.0.1'
```

Example

This op script example demonstrates the output of **jcs:parse-ip**() for multiple IPv4 and IPv6 addresses.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $address-set := {
            <address> "10.0.0.1/255.0.0.0";
            <address> "192.168.254.13/23";
            <address> "192.168.1.1/32";
            <address> "::ffff:0:0:10.100.1.1/96";
            <address> "fc00::1234:AB12/48";
        }
```

```
        /* Go through each address in the set and display its information */
        for-each( $address-set/address ) {
            <output> "Address: " _ .;
            var $result = jcs:parse-ip( . );
            <output> jcs:printf( "%8s %s", "Host:", $result[1]);
            <output> jcs:printf( "%8s %s", "Family:", $result[2]);
            <output> jcs:printf( "%8s %s", "Prefix:", $result[3]);
            <output> jcs:printf( "%8s %s", "Network:", $result[4]);
            if( $result[2] == "inet" ) {
                <output> jcs:printf( "%8s %s", "Mask:", $result[5]);
            }
        }
    }
}
```

Output

```
Address: 10.0.0.1/255.0.0.0
   Host: 10.0.0.1
 Family: inet
 Prefix: 8
Network: 10.0.0.0
   Mask: 255.0.0.0
Address: 192.168.254.13/23
   Host: 192.168.254.13
 Family: inet
 Prefix: 23
Network: 192.168.254.0
   Mask: 255.255.254.0
Address: 192.168.1.1/32
   Host: 192.168.1.1
 Family: inet
 Prefix: 32
Network: 192.168.1.1
   Mask: 255.255.255.255
Address: ::ffff:0:0:10.100.1.1/96
   Host: ::ffff:0:0:a64:101
 Family: inet6
 Prefix: 96
Network: 0:0:0:ffff::
Address: fc00::1234:AB12/48
   Host: fc00::1234:ab12
 Family: inet6
 Prefix: 48
Network: fc00::
```

jcs:printf()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

string **jcs:printf**(string *[format]*)

string **jcs:printf**(string *[format]*, string* *[arguments]*)

Description

The **jcs:printf**() function returns a formatted string that is created by inserting string arguments into a format string. The returned string can then be displayed through a standard output function such as **jcs:output**(), because **jcs:printf**() only creates the formatted string; it does not display it.

The first argument is the format string, followed by a variable number of string arguments, which are inserted into the format string, according to its included format specifications. The format string consists of plain text as well as format specifications. Plain text is copied directly from the format string to the returned formatted string. Format specifications begin with '%' and end with 's' and are used to insert string arguments into the returned string.

Function call:

```
jcs:printf( "First: %s Second: %s", "1st", "2nd" )
```

Returned string:

```
"First: 1st Second: 2nd"
```

To include a '%' character in the formatted text, include '%%' in the format string.

Function call:

```
jcs:printf( "%s%%", "50" )
```

Returned string:

```
"50%"
```

Format specifications consist of optional flags, width, and precision; and are terminated by a mandatory conversion specifier, which should be set to 's' because only string arguments are supported, but any of the following characters will terminate the format specification as well: 'd', 'e', 'E', 'f', 'g', 'G", 'i', 'o', 'u', 'x'; however, these alternate characters are handled the same as 's' (because only strings are supported), so it is best to always terminate format specifications with 's' to prevent confusion with script writers that are accustomed to the printf function in other programming languages.

Each format specification corresponds to a string argument, so there must be at least as many string arguments as there are format specifications (the below sections on width and precision include scenarios where there will be more arguments than format specifications), and the arguments are inserted in the order that the format specifications appear in the format string.

Multiple flags can be used within the same format specification. These are the supported flags:

- '-' - The minus sign flag causes the string to be left-aligned. Alignment determines whether padding is appended or prepended to the string. A right-aligned string, which is the default, has padding prepended, but a left-aligned string, which is indicated by using the '-' flag, has padding appended.
  - Alignment has no effect on string truncation. Right truncation is always performed when a specified precision forces the string to be truncated.

  Function call:

  ```
  jcs:printf( "|%-10s|", "Left" )
  ```

  Returned string:

  ```
  "|Left      |"
  ```

- '0' – A zero flag causes the string to be padded by zeros rather than spaces, but it is only valid for right-aligned strings and is ignored if the '-' flag is present.

  Function call:

  ```
  jcs:printf( "%08s", "5" )
  ```

  Returned string:

  ```
  "00000005"
  ```

- 'j1' – The j1 flag is used to indicate that a string argument should not be inserted if the preceding call to **jcs:printf**() used the same format string, and the string argument's value has not changed. This flag is useful if a duplicate column value should not be displayed in subsequent rows.

  Function call:

  ```
  jcs:printf( "%j1-10s %s", "xe-0/0/0", "inet" )
  jcs:printf( "%j1-10s %s", "xe-0/0/0", "inet6" )
  jcs:printf( "%j1-10s %s", "xe-1/0/0", "inet" )
  ```

  Returned string:

  ```
  "xe-0/0/0   inet"
  "           inet6"
  "xe-1/0/0   inet"
  ```

- 'jc' - The jc flag causes the first letter of the string argument to be capitalized.
  - The first character of the string is capitalized following minimum width padding, but before tag prepending, so capitalization will not work for right-aligned strings that have spaces or zeros prepended due to their minimum width.

  Function call:

  ```
  jcs:printf( "%jcs", "example" )
  ```

  Returned string:

  ```
  "Example"
  ```

- 'jt{TAG}' – The jt{TAG} flag is used to prepend a tag string to the inserted argument string if the argument string is not blank. The tag string consists of all the characters inside the { } curly brackets within the flag.
  - A '}' cannot be included within the tag string, because it terminates the tag string.
  - If the argument is a node-set or a result tree fragment that has element nodes, but no text content then it will be converted into a blank string, so the tag string will not be added.
  - The tag is prepended after the string argument has been formatted to the correct width and precision, which could cause columns to not align correctly, so this flag should only be used within the last format specification of a row if column alignment must be maintained over multiple rows.

  Function call:

  ```
  jcs:printf( "%-10s %jt{>>}12s", "10.0.0.1", "192.168.1.1" )
  jcs:printf( "%-10s %jt{>>}12s", "10.0.0.1", "" )
  ```

  Returned string:

  ```
  "10.0.0.1   >> 192.168.1.1"
  "10.0.0.1              "
  ```

The minimum width, to which a string will be padded if necessary, can be indicated by including a numeric value within the format specification, or a * can be included, indicating that the width should be taken from the argument list rather than from the format specification; this causes the next argument in the argument list to be converted into an integer, which is used as the width, and the following argument is used as the string argument that is inserted into the string.

Function call:

```
jcs:printf( "|%10s|%-10s|", "Right", "Left" )
jcs:printf( "|%*s|%-*s|", 5, "1", 3, "2" )
```

Returned string:

```
"|     Right|Left      |"
"|    1|2  |"
```

The precision of the string, or the maximum length, is indicated by including a period followed by a numeric value within the format specification. If the string argument is longer than the indicated precision then the string will be right-truncated. A * can be included, instead of a number, to indicate that the precision should be taken from the argument list rather than from the format specification. This causes the next argument in the argument list to be converted into an integer, which is used as the precision, and then the following field is used as the string argument that is inserted into the string.

Function call:

```
jcs:printf( "|%5.5s|%.4s|", "1234567890", "abcdefg" )
jcs:printf( "|%*.*s|", 5, 1, "12345" )
```

Returned string:

```
"|12345|abcd|"
"|    1|"
```

The following escape characters can be used:

- \n – Newline
- \r – Carriage Return
- \t - Tab
- \\ - Backslash (As of Junos 10.2)
- \" – Double-quote (As of Junos 10.1R2)
- \' – Single-quote

Example

This op script demonstrates how to create formatted output with the **jcs:printf**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
```

```
        /* Display the parameters */
        expr jcs:output( jcs:printf( "%15s %-10s", "Parameter", "Value" ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$user", $user ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$hostname", $hostname ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$product", $product ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$script", $script ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$localtime", $localtime ) );
        expr jcs:output( jcs:printf( "%15s %-10s", "$localtime-iso", $localtime-iso ) );

        /* Retrieve string to display */
        var $string = jcs:get-input( "Enter string: " );

        /* Retrieve width */
        var $width = jcs:get-input( "Enter width: " );

        /* Retrieve precision */
        var $precision = jcs:get-input( "Enter precision: " );

        expr jcs:output( jcs:printf( "|%*.*s|", $width, $precision, $string ) );
    }
}
```

Output

```
        Parameter Value
            $user jnpr
        $hostname srx210
         $product srx210h
          $script jcs_printf.slax
       $localtime Wed Jan 26 23:10:15 2011
 $localtime-iso 2011-01-26 23:10:15 UTC
Enter string: 1234567890
Enter width: 5
Enter precision: 3
|  123|
```

jcs:progress()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set *[empty]* **jcs:progress**(string)

node-set *[empty]* **jcs:progress**(string, string*)

Description

The **jcs:progress**() function facilitates debugging by displaying a progress message to the CLI user when the script is run with the “detail” option. When the script is run normally the progress message is not displayed,

but the message is always written to the trace file for the particular script type.

jcs:progress() can be called with only a single string argument, or multiple arguments can be provided, causing them to be concatenated into one combined string. Whether written to the trace file or displayed on the screen, a progress message always includes a preceding timestamp, and a newline terminates the output text, so each call to **jcs:progress**() occurs on a separate line.

```
2011-01-29 14:58:51 UTC: Example progress message
```

While all script types are supported, only op and commit scripts can display the progress message to the CLI user; however, all script types write the message to their trace file.

To view progress messages from op scripts, execute the script with the detail option:

```
jnpr@junos> op script-example detail
```

To view progress messages from commit scripts, apply the | display detail option to the commit command:

```
[edit]
jnpr@junos# commit | display detail
```

While **jcs:progress**() does return a node-set, it is always empty and can be ignored, so **jcs:progress**() is normally called with the **expr** statement, rather than assigning its result to a variable:

```
expr jcs:progress( "This is an example" );
The following escape characters can be used:
```

- \n – Newline
- \r – Carriage Return
- \t – Tab
- \\ - Backslash (As of Junos 10.2)
- \" – Double-quote (As of Junos 10.1R2)
- \' – Single-quote

As of Junos 10.2, the maximum length for the output text is 10 KB, and longer strings are truncated to the supported length.

Note: As of the time of this writing, a double-quote and a single-quote cannot both appear in the same string.

Example

This op script example demonstrates how to display progress messages with **jcs:progress**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        expr jcs:progress( "Gathering input" );
        var $input = jcs:get-input( "Enter a string: " );
```

```
            expr jcs:progress( "Converting to uppercase" );
            var $converted = translate( $input, "abcdefghijklmnopqrstuvwxyz",
                "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

            expr jcs:progress( "Original string: ", $input, " Converted string: ", $converted );
            expr jcs:output( "Converted string: ", $converted );
        }
    }
```

Output

```
    jnpr@srx210> op jcs_progress
    Enter a string: test
    Converted string: TEST

    jnpr@srx210> op jcs_progress detail
    2011-01-29 15:08:24 UTC: running op script 'jcs_progress.slax'
    2011-01-29 15:08:24 UTC: opening op script '/var/db/scripts/op/jcs_progress.slax'
    2011-01-29 15:08:24 UTC: reading op script 'jcs_progress.slax'
    2011-01-29 15:08:24 UTC: Gathering input
    Enter a string: example string
    2011-01-29 15:08:26 UTC: Converting to uppercase
    2011-01-29 15:08:26 UTC: Original string: example string Converted string: EXAMPLE STRING
    Converted string: EXAMPLE STRING
    2011-01-29 15:08:26 UTC: inspecting op output 'jcs_progress.slax'
    2011-01-29 15:08:26 UTC: finished op script 'jcs_progress.slax'

    jnpr@srx210> show log op-script.log
    Jan 29 15:08:22 srx210 clear-log[1211]: logfile cleared
    Jan 29 15:08:23 complete script processing begins
    Jan 29 15:08:23 opening op script '/var/db/scripts/op/jcs_progress.slax'
    Jan 29 15:08:23 reading op script 'jcs_progress.slax'
    Jan 29 15:08:24 op script processing begins
    Jan 29 15:08:24 running op script 'jcs_progress.slax'
    Jan 29 15:08:24 opening op script '/var/db/scripts/op/jcs_progress.slax'
    Jan 29 15:08:24 reading op script 'jcs_progress.slax'
    Jan 29 15:08:24 Gathering input
    Jan 29 15:08:26 Converting to uppercase
    Jan 29 15:08:26 Original string: example string Converted string: EXAMPLE STRING
    Jan 29 15:08:26 op script output
    Jan 29 15:08:26 begin dump
    <?xml version="1.0"?>
    <op-script-results xmlns:junos="http://xml.juniper.net/junos/*/junos" xmlns:xnm="http://xml.
juniper.net/xnm/1.1/xnm" xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"/>
    Jan 29 15:08:26 end dump
    Jan 29 15:08:26 inspecting op output 'jcs_progress.slax'
    Jan 29 15:08:26 finished op script 'jcs_progress.slax'
    Jan 29 15:08:26 op script processing ends
```

jcs:regex()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set *[matches]* **jcs:regex**(string *[pattern]*, string *[target]*)

Description

The **jcs:regex**() function is used to match strings based on a regular expression. The first string argument is the regular expression pattern in POSIX extended regular expression format, and the second string argument is the target string that is searched for a match. A node-set is returned, which contains the string that matches the entire regular expression, as well as up to eight included subexpressions.

Supported regular expression operators:

- Match any character - "." – Matches any character, including newlines.

  Function call:

  ```
  jcs:regex( "srx2.0", "srx210" )
  ```

  Returned node-set:

  ```
  <match> "srx210"
  ```

- Match zero or more – "*" – Matches the prior character or subexpression zero or more times. Function call:

  ```
  jcs:regex( "srx.*", "srx210" )
  ```

  Returned node-set:

  ```
  <match> "srx210"
  ```

- Match one or more – "+" – Matches the prior character or subexpression one or more times. Function call:

  ```
  jcs:regex( "s+rx210", "srx210" )
  ```

  Returned node-set:

  ```
  <match> "srx210"
  ```

- Match zero or one times – "?" – Matches the prior character or subexpression either zero or one times.

  Function call:

  ```
  jcs:regex( "srx21?0", "srx210" )
  ```

  Returned node-set:

  ```
  <match> "srx210"
  ```

- Matching interval – "{ ... }" – Comes in three forms:
  - Match exactly N times: {N}
  - Match N times or more: {N,}
  - Match between N1 and N2 times: {N1,N2}
  - N/N1/N2 must be 255 or less

  Function call:

  ```
  jcs:regex( "srx[0-9]{3}", "srx210" )
  ```

  Returned node-set:

  ```
  <match> "srx210"
  ```

- Alternation – "|" – Matches one or the other regular expressions. The longer left-most match is preferred.

Function call:

```
jcs:regex( "20|2011", "2011-02-22" )
```

Returned node-set:

```
<match> "2011"
```

- Matching list – "[...]" – Matches one character from within the list.
  - Ranges are expressed by using the hyphen:

Function call:

```
jcs:regex( "srx[0-9]*", "srx210" )
```

Returned node-set:

```
<match> "srx210"
```

  - Most special characters within a list are treated as normal characters (*, +, ?, etc.) and do not require escaping.
  - To include a hyphen "-" as a normal character, include it as the first or last character.
  - To include a closing bracket "]" as a normal character, include it as the first character.
- Non-matching list – "[^ ...]" – Matches any character that is not included in the list. The syntax rules are the same as a matching list.

Function call:

```
jcs:regex( "[^ 0-9]*", "JUNOS 10.4R1.9" )
```

Returned node-set:

```
<match> "JUNOS"
```

- Grouping – "(...)" – Creates a group or subexpression, allowing operators to be applied to groups rather than characters, and returns the matching value as part of the returned node-set.

Function call:

```
jcs:regex( "([0-9.]*).([0-9.]*)", "10.4R1.9" )
```

Returned node-set:

```
<match> "10.4R1.9";
<match> "10.4";
<match> "1.9";
```

Supported regular expression anchors:

- Start of string – "^" – Anchors the regular expression to the start of the string.

Function call:

```
jcs:regex( "^[^ ]*", "JUNOS Software Release [10.4R1.9]" )
```

Returned node-set:

```
<match> "JUNOS"
```

- End of string – "$" – Anchors the regular expression to the end of the string.

Function call:

```
jcs:regex( "[^ ]*$", "JUNOS Software Release [10.4R1.9]" )
```

Returned node-set:

```
<match> "[10.4R1.9]"
```

- Start of word –"[[:<:]]" – Anchors the regular expression to the start of a word, where a word is a sequence of alphanumeric characters or underscores.

Function call:

```
jcs:regex( "[[:<:]]S[^ ]*", "JUNOS Software Release [10.4R1.9]" )
```

Returned node-set:

```
<match> "Software"
```

- End of word –"[[:>:]]" – Anchors the regular expression to the end of a word, where a word is a sequence of alphanumeric characters or underscores.

Function call:

```
jcs:regex( ".*S[[:>:]]", "JUNOS Software Release [10.4R1.9]" )
```

Returned node-set:

```
<match> "JUNOS"
```

Escaping:

- Most standard escape characters are expressed in the normal fashion: \n \t \r \" \' .
  - Exception is \\ which must be expressed as \\\\ within a regular expression, except within a list. This is necessary because the escape must be present both for the SLAX to XSLT conversion as well as for the **jcs:regex**() processing. In other words, the SLAX to XSLT conversion translates "\\\\" to "\\", which **jcs:regex**() treats correctly as an escaped backslash.

Function call:

```
jcs:regex( "1\\\\2", "1\\2" )
```

Returned node-set:

```
<match> "1\2"
```

  - Double-quote escape "\" supported as of Junos 10.1R2
- Regular expression special characters (operators and anchors) must be escaped with two backslashes, because the SLAX to XSLT conversion will remove one of them.

Function call:

```
jcs:regex( "\\.\\^\\$\\*\\+\\(\\)\\[\\]\\{\\}\\|\\?", ".^$*+()[]{}|?" )
```

Returned node-set:

```
<match> ".^$*+()[]{}|?"
```

Character classes can be used within a list to indicate a class of characters that should be included instead of specifying the individual characters. The following character classes are supported by **jcs:regex**():

- [:alnum:] – Alphanumeric characters: A-Z, a-z, and 0-9
- [:alpha:] – Alpha characters: A-Z, and a-z
- [:blank:] – Blank characters: space and tab
- [:cntrl:] – Control characters: ASCII values 0x0-0x19 and 0x7F
- [:digit:] – Numeric characters: 0-9
- [:graph:] – Printable characters, except for space: ASCII values 0x21-0x7E
- [:lower:] – Lowercase letters: a-z
- [:print:] – Printable characters: ASCII values 0x20-0x7E
- [:punct:] – Punctuation: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
- [:space:] – Whitespace: space, tab, newline, carriage return, vertical tab, form feed

- [:upper:] – Uppercase letters: A-Z
- [:xdigit:] – Hexadecimal digit characters: A-F, a-f, 0-9

Function call:

```
jcs:regex( "[[:alnum:][:punct:]]*", "*[Direct/0] 05:08:47" )
```

Returned node-set:

```
<match> "*[Direct/0]"
```

While no multi-character collating sequences are supported by **jcs:regex**(), single character collating elements can be used within lists and could be used to represent special characters or to indicate a specific control character. Collating elements for each non-null ASCII character can be created by enclosing the character within "[." and ".]". For example, to match a "]" within a list, the collating sequence "[.].]" could be used:

Function call:

```
jcs:regex( "[[[:digit:][.].]]*", "[10]" )
```

Returned node-set:

```
<match> "[10]"
```

In addition, the following collating sequences are defined:

Sequence	ASCII	Sequence	ASCII	Sequence	ASCII
[.SOH.]	0x01	[.SUB.]	0x1A	[.three.]	0x33 "3"
[.STX.]	0x02	[.ESC.]	0x1B	[.four.]	0x34 "4"
[.ETX.]	0x03	[.IS4.] [.FS.]	0x1C	[.five.]	0x35 "5"
[.EOT.]	0x04	[.IS3.] [.GS.]	0x1D	[.six.]	0x36 "6"
[.ENQ.]	0x05	[.IS2.] [.RS.]	0x1E	[.seven.]	0x37 "7"
[.ACK.]	0x06	[.IS1.] [.US.]	0x1F	[.eight.]	0x38 "8"
[.BEL.] [.alert.]	0x07	[.space.]	0x20 " "	[.nine.]	0x39 "9"
[.BS.] [.backspace.]	0x08	[.exclamation-mark.]	0x21 "!"	[.colon.]	0x3A ":"
[.HT.] [.tab.]	0x09	[.quotation-mark.]	0x22 "	[.semicolon.]	0x3B ";"
[.LF.] [.newline.]	0x0A	[.number-sign.]	0x23 "#"	[.less-than-sign.]	0x3C "<"
[.VT.] [.vertical-tab.]	0x0B	[.dollar-sign.]	0x24 "$"	[.equals-sign.]	0x3D "="

[.FF.] [.form-feed.]	0x0C	[.percent-sign.]	0x25 "%"	[.greater-than-sign.]	0x3E ">"
[.CR.] [.carriage-return.]	0x0D	[.ampersand.]	0x26 "&"	[.question-mark.]	0x3F "?"
[.SO.]	0x0E	[.apostrophe.]	0x27 '	[.commercial-at.]	0x40 "@"
[.SI.]	0x0F	[.left-parenthesis.]	0x28 "("	[.left-square-bracket.]	0x5B "["
[.DLE.]	0x10	[.right-parenthesis.]	0x29 ")"	[.backslash.] [.reverse-solidus.]	0x5C "\"
[.DC1.]	0x11	[.asterisk.]	0x2A "*"	[.right-square-bracket.]	0x5D "]"
[.DC2.]	0x12	[.plus-sign.]	0x2B "+"	[.circumflex.] [.circumflex-accent.]	0x5E "^"
[.DC3.]	0x13	[.comma.]	0x2C ","	[.underscore.] [.low-line.]	0x5F "_"
[.DC4.]	0x14	[.hyphen.] [.hyphen-minus.]	0x2D "-"	[.grave-accent.]	0x60 "`"
[.NAK.]	0x15	[.period.] [.full-stop.]	0x2E "."	[.left-brace.] [.left-curly-bracket.]	0x7B "{"
[.SYN.]	0x16	[.slash.] [.solidus.]	0x2F "/"	[.vertical-line.]	0x7C "|"
[.ETB.]	0x17	[.zero.]	0x30 "0"	[.right-brace.] [.right-curly-bracket.]	0x7D "}"
[.CAN.]	0x18	[.one.]	0x31 "1"	[.tilde.]	0x7E "~"
[.EM.]	0x19	[.two.]	0x32 "2"	[.DEL.]	0x7F

Function call:

```
jcs:regex( "[[:digit:][.period.]]*", "10.4R1.9" )
```

Returned node-set:

```
<match> "10.4"
```

There are no defined character equivalence classes so enclosing collating elements in "[=" and "=]" has the same effect as enclosing them in "[." and ".]".

The node-set returned by **jcs:regex**() consists of <match> element nodes with the appropriate match assigned as the text content. If no match is found then an empty node-set is returned. If there is an error with the regular expression then an error message is displayed and an empty node-set is returned. The order of the nodes within the returned node-set is deterministic, so the node-set can be treated similar to an array in other programming languages, meaning that the nodes can be retrieved based on their numerical order. Node

number 1 is always the match of the entire regular expression, and nodes 2 through 9 contain subexpression matches, if appropriate. The subexpression nodes occur within the node-set in the same order as they appear in the regular expression pattern, but they are only included if they have a match, or if they do not have a match but a latter subexpression has a match, in which case they are included with an empty string as their text contents.

Examples of returned node-sets:

Function call:

```
jcs:regex( "123*", "other" )
```

Returned node-set:

```
Empty
```

Function call:

```
jcs:regex( "[A-Z]*", "JUNOS 10.4R1.9" )
```

Returned node-set:

```
<match> "JUNOS"
```

Function call:

```
jcs:regex( "([0-9]{4})-([0-9]{2})-([0-9]{2})", "2010-12-04" )
```

Returned node-set:

```
<match> "2010-12-04"
<match> "2010"
<match> "12"
<match> "04"
```

Function call:

```
jcs:regex( "(1?)(2?)(3?)(4?)(5)(6)(7)(8)(9)", "56789" )
```

Returned node-set:

```
<match> "56789"
<match> ""
<match> ""
<match> ""
<match> ""
<match> "5"
<match> "6"
<match> "7"
<match> "8"
```

Regular expression features not discussed above should be considered unsupported, in particular the following are either not supported or do not work at the time of this writing: backreferences, shorthand character classes, buffer operators, lookaround, non-capturing groups, and non-greedy (lazy) repetition.

Example

This op script demonstrates how to use **jcs:regex**() to extract specific substrings from within a larger string.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
    import "../import/junos.xsl";

    match / {
        <op-script-results> {

            /* parse the $localtime-iso parameter */
            var $regex =
            "([[:digit:]]*)-0?([[:digit:]]*)-0?([[:digit:]]*) 0?([0-9]*):0?([0-9]*):0?([0-
9]*).*";
            var $result = jcs:regex($regex, $localtime-iso );

            /* Display the complete match */
            <output> "Time: " _ $result[1];

            /* Display all the captured subexpressions */
            <output> "Year: " _ $result[2];
            <output> "Month: " _ $result[3];
            <output> "Day: " _ $result[4];
            <output> "Hour: " _ $result[5];
            <output> "Minute: " _ $result[6];
            <output> "Second: " _ $result[7];
        }
    }
```

Output

```
Time: 2011-02-23 16:14:06 UTC
Year: 2011
Month: 2
Day: 23
Hour: 16
Minute: 14
Second: 6
```

jcs:sleep()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

string *[blank]* **jcs:sleep**(string *[seconds]*)

string *[blank]* **jcs:sleep**(string *[seconds]*, string *[milliseconds]*)

Description

The **jcs:sleep**() function causes script processing to pause for the specified number of seconds, expressed in the first argument, plus the indicated number of milliseconds, which is optionally expressed in the second argument. If both seconds and milliseconds are provided then their values are combined to determine the pause length.

While **jcs:sleep**() returns a string, it is always blank and can be ignored.

The argument data types are strings rather than numbers, because it is possible to input the seconds and milliseconds values in decimal, hexadecimal ("0x" prefix), or octal ("0" prefix):

```
/* Decimal - 1 second */
expr jcs:sleep( "1" );
/* Hexadecimal - 10 seconds + 2000 milliseconds */
expr jcs:sleep( "0xA", "0x7D0" );
/* Octal - 13 seconds */
expr jcs:sleep( "015" );
```

However, the arguments are typically provided in decimal, so it is more common to use a number data type for the function's argument, rather than a string, but due to automatic type conversion, there is no difference between either approach:

```
/* Sleep for 500 milliseconds */
expr jcs:sleep( 0, 500 );
```

Example

This op script example demonstrates how to use the **jcs:sleep**() function to temporarily pause script processing.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns date = "http://exslt.org/dates-and-times";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get number of seconds to sleep */
        var $seconds = jcs:get-input( "How many seconds?: " );

        /* Display current time */
        <output> date:time();

        /* Sleep */
        expr jcs:sleep( $seconds );

        /* Display time now */
        <output> date:time();
    }
}
```

Output

```
How many seconds?: 5
23:18:11
23:18:16
```

jcs:split()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.4

Syntax

node-set *[substrings]* **jcs:split(** string *[pattern]*, string *[target]*)

node-set *[substrings]* **jcs:split(** string *[pattern]*, string *[target]* , number *[limit]*)

Description

The **jcs:split**() function splits a target string into multiple substrings based on the location of a regular expression pattern. The first argument is the regular expression pattern, expressed in POSIX extended regular expression format. (For details on the regular expression syntax supported by **jcs:split**(), refer to the **jcs:regex**() function description). The second argument is the target string to be split, and the third argument is an optional limit to the number of returned substrings. If a limit is set then the final substring contains the remaining undivided portion of the target string.

The string is split by searching for a match to the regular expression pattern within the target string and then adding the substring that occurs prior to that match into the returned node-set. The remainder of the string, following the regular expression match, is then searched for another match. If no more matches can be found, then a final substring, consisting of the remainder of the string, is added to the returned node-set. Because the regular expression is evaluated each time the string is split, it is possible to have the split based on a different delimiter each time. If the regular expression match occurs at the beginning or the end of the string then an empty substring is added to the node-set at the appropriate location.

The node-set returned by **jcs:split**() consists of <split> element nodes with the appropriate substring assigned as the text content. The order of the nodes within the node-set is deterministic, so it can be treated similar to an array in other programming languages, with the first node corresponding to the left-most substring from the target string, etc.

If no match is found then the entire target string is returned in a single <split> node. If there is an error with the regular expression then an error message is displayed and an empty node-set is returned.

Examples of returned node-sets:

Function call:

```
jcs:split( "[[:alpha:]]+", "1abc2x3tal4we5" )
```

Returned node-set:

```
<split> "1"
<split> "2"
<split> "3"
<split> "4"
<split> "5"
```

Function call:

```
jcs:split( "[[:digit:]]+", "1abc2x3tal4we5" )
```

Returned node-set:

```
<split> ""
<split> "abc"
<split> "x"
<split> "tal"
<split> "we"
<split> ""
```

Function call:

```
jcs:split( ",", "a,b,c,d,e,f,g", 4 )
```

Returned node-set:

```
<split> "a"
<split> "b"
<split> "c"
<split> "d,e,f,g"
```

Example

This op script demonstrates how to use **jcs:split()** to divide multiple strings.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        call show-substrings( $pattern = "[A-Z]+", $string = "10.4R1.9" );

        call show-substrings( $pattern = "\\.", $string = "10.100.2.1" );

        call show-substrings( $pattern = "[[.space.]]+", $string =
            "ge-0/0/0.0              up    up   inet     10.0.0.10/24");
    }
}

template show-substrings( $pattern, $string ) {
    expr jcs:output( "_\r \n", $string );
    expr jcs:output( str:padding( string-length( $string ), "-" ) );
    var $result = jcs:split( $pattern, $string );
    for-each( $result ) {
        expr jcs:output( . );
    }
}
```

Output

```
10.4R1.9
--------
10.4
1.9

10.100.2.1
----------
```

```
10
100
2
1

ge-0/0/0.0              up    up   inet     10.0.0.10/24
------------------------------------------------------
ge-0/0/0.0
up
up
inet
10.0.0.10/24
```

jcs:sysctl()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

string *[value]* **jcs:sysctl**(string *[name]*)

string *[value]* **jcs:sysctl**(string *[name]*, string *[type]*)

Description

The **jcs:sysctl**() function retrieves a single string or integer sysctl value, which is specified in the first argument. A second optional argument can be set to either "s" or "i" to indicate that the value is either a string or an integer, respectively. If a type is not specified, or if it is anything other than "i", then the type defaults to string.

To see the available sysctl values, run "sysctl -a" from the Junos shell. **jcs:sysctl**() only reads sysctl values. It does not set them. Only string or integer sysctl values are supported, and the type must be specified correctly or the value will not be returned successfully. An invalid sysctl value causes the script to halt and an error to be displayed:

```
error: sysctl error: No such file or directory
```

Example

This op script demonstrates multiple calls to **jcs:sysctl**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";
```

```
match / {
    <op-script-results> {

        <output> "kern.osrelease=" _ jcs:sysctl("kern.osrelease");
        <output> "kern.hostname=" _ jcs:sysctl("kern.hostname", "s");
        <output> "kern.osrevision=" _ jcs:sysctl("kern.osrevision", "i");
        <output> "hw.re.mastership=" _ jcs:sysctl("hw.re.mastership", "i");
        <output> "hw.product.model=" _ jcs:sysctl("hw.product.model", "s");
    }
}
```

Output

```
kern.osrelease=10.4R1.9
kern.hostname=srx210
kern.osrevision=199506
hw.re.mastership=1
hw.product.model=srx210h
```

jcs:syslog()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.0

Syntax

node-set *[empty]* **jcs:syslog**(string *[priority]*, string *[message]*)

node-set *[empty]* **jcs:syslog**(string *[priority]*, string *[message]*, string* *[messages]*)

Description

The **jcs:syslog()** function logs a message to the syslog with the indicated priority (facility and severity). The first argument is the priority, which can be expressed as a string in "facility.severity" form, or as a priority number that is computed based on the facility and severity. The second argument is the message that should be logged. If more than two arguments are present then all arguments besides the priority are concatenated together into a single message string.

Supported syslog facilities:

Facility String	Priority Base	Description
`auth`	32	Authorization system
`change`	176	Configuration change log
`conflict`	168	Configuration conflict log
`daemon`	24	Various system processes
`dfc`	136	Dynamic flow capture

external	144	Local external applications
firewall	152	Firewall filtering system
ftp	88	FTP process
interact	184	Commands executed via CLI
ntp	96	NTP process
pfe	160	Packet forwarding engine
security	104	Security related
user	8	User processes

Supported syslog severities:

Severity String	Value	Description
alert	1	Conditions that require immediate correction
crit	2	Critical conditions
debug	7	Debug messages
emerg panic	0	Panic conditions
err error	3	Error conditions
info	6	Informational messages
notice	5	Non-error conditions that require special handling
warn warning	4	Warning messages

The numeric priority value is computed by adding the severity's value to the facilities priority base. For example, a priority of 10 would log to the user facility with a critical severity. Here is another example of specifying the priority through its numeric value:

```
/* Facility: external Severity: alert */
expr jcs:syslog( 145, "Alert! Alert!" );
```

Alternatively, the priority can be specified by including the authorization string, a period, and then the severity string. Here is an example of specifying the priority through the string values:

```
/* Facility: authorization Severity: info */
expr jcs:syslog( "auth.info", "This is an informational message" );
```

All syslog messages that are generated by scripts have the string "cscript:" prepended to their text. For example, here are the two above syslog messages, as they would appear in the syslog file:

```
Feb 26 23:31:49  srx210 cscript: This is an informational message
Feb 26 23:31:49  srx210 cscript: Alert! Alert!
```

All script types can use **jcs:syslog**() to write syslog messages; however, commit scripts have less of a need to do so given that they have access to the <syslog> result tree element, which can also log messages to the syslog.

The following escape characters can be used:

- \t – Tab

- \\ - Backslash (As of Junos 10.2)
- \" – Double-quote (As of Junos 10.1R2)
- \' – Single-quote

While **jcs:syslog**() does return a node-set, it is always empty and can be ignored, so **jcs:syslog**() is normally called with the **expr** statement, rather than assigning its result to a variable:

```
expr jcs:syslog( "Example log message" );
```

If there is an error with the facility or severity then an error message is displayed similar to this:

```
error: syslog error: Invalid facility number
```

Example

This op script demonstrates how **jcs:syslog**() can log messages to the syslog at different priorities.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        expr jcs:syslog( "external.info", "Example syslog message: \"Insert message here\"" );
        expr jcs:syslog( "daemon.debug", "This message is from ", $script );
        expr jcs:syslog( 12, "This is a ", "user\\", "warning." );
    }
}
```

Output

```
(With explicit-priority configured for the syslog file):
Feb 26 23:58:00  srx210 cscript: %EXTERNAL-6: Example syslog message: "Insert message here"
Feb 26 23:58:00  srx210 cscript: %DAEMON-7: This message is from jcs_syslog.slax
Feb 26 23:58:00  srx210 cscript: %USER-4: This is a user\warning.
```

jcs:trace()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set *[empty]* **jcs:trace**(string)

node-set *[empty]* **jcs:trace**(string, string*)

Description

The **jcs:trace**() function logs a message to the appropriate trace file for the script type. It can be called with only a single string argument, or multiple arguments can be provided, causing them to be concatenated into one combined string. A trace message always includes a preceding timestamp, and a newline terminates the output text, so each call to **jcs:trace**() occurs on a separate line.

```
Feb 28 12:23:06 Example trace
```

While **jcs:trace**() does return a node-set, it is always empty and can be ignored, so **jcs:trace**() is normally called with the **expr** statement, rather than assigning its result to a variable:

```
expr jcs:trace( "This is an example" );
```

The following escape characters can be used:

- \n – Newline
- \r – Carriage Return
- \t – Tab
- \\ - Backslash (As of Junos 10.2)
- \" – Double-quote (As of Junos 10.1R2)
- \' – Single-quote

As of Junos 10.2, the maximum length for the output text is 10 KB, and longer strings are truncated to the supported length.

Note: As of the time of this writing, a double-quote and a single-quote cannot both appear in the same string.

Example

This op script example demonstrates how to log trace messages with **jcs:trace**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        expr jcs:trace( "Script: ", $script );
        expr jcs:trace( "User: ", $user );
        expr jcs:trace( "Time: ", $localtime-iso );

    }
}
```

Output

```
Feb 28 12:12:27 Script: jcs_trace.slax
Feb 28 12:12:27 User: jnpr
Feb 28 12:12:27 Time: 2011-02-28 12:12:27 UTC
```

key()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

node-set **key**(string *[name]*, string *[value]*)

node-set **key**(string *[name]*, node-set *[values]*)

Description

As an alternative to IDs, keys can be used to index the nodes within a XML document. They are defined by including **<xsl:key>** top-level elements within the script, which consist of a name, the nodes to index, and the value that is paired with the key name to reference the matching nodes. The **key**() function is used to retrieve the nodes that are referenced by a particular name and value. It takes two arguments: the string key name and the desired key's value, and it returns all nodes referenced by that key name and value as a node-set. If the desired key value is provided as a node-set, rather than a string, then the returned node-set is a union of all the referenced nodes for the key values expressed by the nodes within the node-set.

Similar to the **id**() function, the **key**() function works with the XML document of the current node, so the current node might have to be explicitly changed, by using a **for-each** loop for example, in order to retrieve nodes from the desired XML document.

Example

This op script example indexes the XML results of the <get-route-information> Junos XML API element according to their protocol and next-hop. It shows how to use the **key**() function to retrieve all nodes that are indexed according to a specific key name and value.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Define <xsl:key> elements */
<xsl:key name="protocol" match="route-table/rt" use="rt-entry/protocol-name">;
```

```
<xsl:key name="next-hop" match="route-table/rt" use="rt-entry/nh/via">;

match / {
    <op-script-results> {
        var $results = jcs:invoke("get-route-information");

        /* Change current node to the $results XML document */
        for-each( $results ) {

            /* Display all static routes */
            <output> "Static routes: ";
            for-each( key( "protocol", "Static" ) ) {
                <output> rt-destination;
            }

            /* Display all routes with next-hop of ge-0/0/0.0 */
            <output> "Next-hop ge-0/0/0.0: ";
            for-each( key( "next-hop", "ge-0/0/0.0" ) ) {
                <output> rt-destination;
            }
        }
    }
}
```

jcs:invoke() results:

```
<route-information>
    <!-- keepalive -->
    <route-table>
        <table-name>inet.0</table-name>
        <destination-count>5</destination-count>
        <total-route-count>5</total-route-count>
        <active-route-count>4</active-route-count>
        <holddown-route-count>0</holddown-route-count>
        <hidden-route-count>1</hidden-route-count>
        <rt junos:style="brief">
            <rt-destination>0.0.0.0/0</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="1883">00:31:23</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.1</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.0/24</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Direct</protocol-name>
                <preference>0</preference>
                <age junos:seconds="1883">00:31:23</age>
                <nh>
                    <selected-next-hop/>
```

```
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.10/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Local</protocol-name>
                <preference>0</preference>
                <age junos:seconds="1887">00:31:27</age>
                <nh-type>Local</nh-type>
                <nh>
                    <nh-local-interface>ge-0/0/0.0</nh-local-interface>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>192.168.255.254/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="651">00:10:51</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.2</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
    </route-table>
</route-information>
```

Output

```
Static routes:
0.0.0.0/0
192.168.255.254/32
Next-hop ge-0/0/0.0:
0.0.0.0/0
10.0.0.0/24
192.168.255.254/32
```

lang()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **lang**(string *[language code]*)

Description

The **lang**() function returns true or false based on whether the provided language code matches the language specified for the context node through the xml:lang attribute, or not. If the xml:lang attribute is not present on the context node then the nearest usage of xml:lang among the node's ancestors will be selected instead.

Junos does not include this attribute in its generated XML so the **lang**() function is not commonly used, but it could be useful when working with a third-party XML document.

The language code is considered a match if it equals the selected xml:lang attribute of the context node regardless of case, or if the xml:lang attribute includes a hyphen to specify a sublanguage and the argument string matches the attribute's value prior to the hyphen.

Example

This is a simple op script example of how the **lang**() function can be used to differentiate between multiple nodes based on their xml:lang attribute values.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $language-example := {
            <hello xml:lang="en"> "hello";
            <hello xml:lang="es"> "hola";
        }

        for-each( $language-example/hello ) {
            if( lang( "en" ) ) {
                <output> . _ " is in english";
            }
            if( lang( "es" ) ) {
                <output> . _ " is in spanish";
            }
        }
    }
}
```

Output

```
hello is in english
hola is in spanish
```

last()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **last**()

Description

The **last**() function returns the current context size. The context size is the number of nodes in the node-set being evaluated by a predicate, or if **last**() is being used outside of a predicate, the context size is the number of nodes in the current node list. The latter usage typically only occurs within a **for-each** loop where the current node list consists of all the nodes that were selected by the for-each statement's location path.

Example

This op script example shows the effect of using the **last**() function in both location path predicates as well as within for-each loops.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $host-name-set := {
        <host-name> "PE1";
        <host-name> "P1";
        <host-name> "P2";
        <host-name> "PE2";
    }

    var $last-host-name = $host-name-set/host-name[ last() ];
    expr jcs:output( "Last host-name: ", $last-host-name );

    var $last-p-host-name = $host-name-set/host-name[not(starts-with(., "PE"))][last()];
    expr jcs:output( "Last P host-name: ", $last-p-host-name );

    for-each( $host-name-set/host-name ) {
        if( position() == last() ) {
            expr jcs:output( "All host-name nodes context size: ", last() );
        }
    }

    for-each( $host-name-set/host-name[ starts-with( ., "PE" )] ) {
        if( position() == last() ) {
```

```
            expr jcs:output( "All PE host-name nodes context size: ", last() );
        }
    }
}
```

Output

```
Last host-name: PE2
Last P host-name: P2
All host-name nodes context size: 4
All PE host-name nodes context size: 2
```

libxslt:node-set()

Source: Libxslt
Namespace: http://xmlsoft.org/XSLT/namespace
Common Prefix: libxslt

Alternate Prefix: ext

Minimum Version: Junos 8.2

Syntax

node-set **libxslt:node-set**(node-set)

node-set **libxslt:node-set**(result-tree-fragment)

Description

The **libxslt:node-set**() function is used to convert a result tree fragment into a node-set. The behavior differs depending on the data type of the argument:

- Result tree fragment - A new XML document is created consisting of the nodes expressed by the result tree fragment. The returned node-set contains the root node of the new XML document.
- Node-set – The node-set argument is returned as the result.

One common stumbling block when using the **libxslt:node-set**() function is remembering that a converted result tree fragment results in a node-set that contains the root node of the new XML document, rather than all of the XML children. For example, the following result tree fragment:

```
var $rtf = {
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/1";
}
```

would be converted into a node-set with a single node: the root-node of the XML document, of which both of the <interface> nodes would be children; however, if a script writer wishes to have the node-set contain all the top-level child nodes rather than the root node then the following code can be used:

```
var $ns = libxslt:node-set( $rtf )/*;
```

The above example first converts the result tree fragment variable into a node-set, the returned root node is then subjected to a location-path to retrieve its children, and it is these child nodes that are assigned as a node-set to the $ns variable.

Note: The **libxslt:node-set**(), **saxon:node-set**(), and **xt:node-set**() functions are all identical. They also provide similar functionality as the **exsl:node-set**() function, except that the latter function is capable of converting strings, booleans, and numbers into node-sets as well.

Example

This op script demonstrates how the various node-set functions can be used both to convert a result tree fragment variable into a node-set as well as a string into a node-set.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns libxslt = "http://xmlsoft.org/XSLT/namespace";
ns saxon = "http://icl.com/saxon";
ns xt = "http://www.jclark.com/xt";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";


match / {
    <op-script-results> {
        var $rtf = {
            <node> "one";
            <node> "two";
            <node> "three";
        }
        /* Convert RTF into node-set using all four methods */
        var $l-ns = libxslt:node-set( $rtf );
        var $s-ns = saxon:node-set( $rtf );
        var $x-ns = xt:node-set( $rtf );
        var $e-ns = exsl:node-set( $rtf );

        /* Display counts */
        <output> "libxslt #" _ count( $l-ns/node );
        <output> "saxon #" _ count( $s-ns/node );
        <output> "xt #" _ count( $x-ns/node );
        <output> "exsl #" _ count( $e-ns/node );

        /* Only exsl:node-set can convert string, booleans, and numbers */
        for-each( exsl:node-set( "Example text" ) ) {
            <output> "String converted by exsl:node-set() into text node: " _ .;
        }
    }
}
```

Output

```
libxslt #3
saxon #3
xt #3
exsl #3
String converted by exsl:node-set() into text node: Example text
```

local-name()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **local-name**()

string **local-name**(node-set)

Description

Node names can consist of both a local part as well as a namespace. The **local-name**() function returns the local part of the node name. If called with no argument then it returns the local name of the context node. Otherwise, it returns the local name of the node in the node-set argument. If multiple nodes are included within the node-set then the first node in document order is selected.

If the provided node has no name, or the node-set argument is empty, then an empty string is returned.

Example

In this op script example an incorrect Junos XML API request is performed so that a <xnm:error> is returned by **jcs:invoke**(). Both the qualified and local name of the <xnm:error> node are displayed by using the **name**() and **local-name**() functions.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $results = jcs:invoke("invalid-rpc");
    expr jcs:output( "Qualified name: ", name( $results ) );
    expr jcs:output( "Local-name: ", local-name( $results ) );
}
```

Output

```
Qualified name: xnm:error
Local-name: error
```

math:abs()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:abs**(number)

Description

The **math:abs**() function returns the absolute value of the number argument.

Example

This op script computes the absolute value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:abs( 100 );
        <output> math:abs( -25 );
        <output> math:abs( 0 );
        <output> math:abs( -5.5 );
    }
}
```

Output

```
100
25
0
5.5
```

math:acos()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:acos**(number)

Description

The **math:acos**() function returns the arccosine radian value of the number argument.

Example

This op script computes the arccosine value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:acos( 1 );
        <output> math:acos( 0 );
        <output> math:acos( -0.5 );
    }
}
```

Output

```
0
1.570796326794897
2.094395102393196
```

math:asin()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:asin**(number)

Description

The **math:asin**() function returns the arcsine radian value of the number argument.

Example

This op script computes the arcsine value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:asin( 1 );
        <output> math:asin( 0 );
        <output> math:asin( -0.5 );
    }
}
```

Output

```
1.570796326794897
0
-0.523598775598299
```

math:atan()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:atan**(number)

Description

The **math:atan**() function returns the arctangent radian value of the number argument.

Example

This op script computes the arctangent value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:atan( 100 );
        <output> math:atan( 0 );
        <output> math:atan( -5000 );
    }
}
```

Output

```
1.560796660108231
0
-1.570596326797563
```

math:atan2()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:atan2**(number *[y]*, number *[x]*)

Description

The **math:atan2**() function returns the angle's radian value from the X axis to a point, where the first number argument is the Y axis of the point and the second number argument is the X axis of the point.

Example

This op script demonstrates the use of the **math:atan2**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:atan2( 1, 5 );
        <output> math:atan2( 2, 3 );
        <output> math:atan2( 10, 20 );
    }
}
```

Output

```
1.373400766945016
0.982793723247329
1.10714871779409
```

math:constant()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:constant**(string *[name]*, number *[precision]*)

Description

The **math:constant**() function returns the constant specified as a string in the first argument. The following constants can be used:

- PI
- E
- SQRRT2
- LN2
- LN10
- LOG2E
- SQRT1_2

The second argument is a number that indicates the precision of the number that should be returned, referring to the total number of characters including the decimal point.

Example

This op script shows how **math:constant**() can be used to return the constant values with a specific precision.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Create converted node-set of the strings */
        var $constants := {
            <constant> "PI";
            <constant> "E";
            <constant> "SQRRT2";
            <constant> "LN2";
            <constant> "LN10";
            <constant> "LOG2E";
            <constant> "SQRT1_2";
        }

        /* Loop through and report constant values with precision of 8 */
        for-each( $constants/constant ) {
            <output> . _ " = " _ math:constant( ., 8 );
        }
    }
}
```

Output

```
PI = 3.141592
E = 2.718281
SQRRT2 = 1.414213
LN2 = 0.693147
LN10 = 2.302585
LOG2E = 1.442695
SQRT1_2 = 0.707106
```

math:cos()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math

Minimum Version: Junos 9.4

Syntax

number **math:cos**(number)

Description

The **math:cos**() function returns the cosine radian value of the number argument.

Example

This op script computes the cosine value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:cos( 1 );
        <output> math:cos( 0 );
        <output> math:cos( -0.5 );
    }
}
```

Output

```
0.54030230586814
1
0.877582561890373
```

math:exp()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:exp**(number)

Description

The **math:exp**() function returns the value of e (Euler's number, the base of natural logarithms) by the power expressed by the number argument.

Example

This op script displays some results of **math:exp**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:exp( 1 );
        <output> math:exp( 0 );
        <output> math:exp( 5 );
    }
}
```

Output

```
2.718281828459046
1
148.413159102577
```

math:highest()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

node-set **math:highest**(node-set *[numbers]*)

Description

The **math:highest**() function returns the node in the node-set that contains the maximum numeric value. If multiple nodes share the same maximum value then they are all returned. If any nodes evaluate to NaN then the returned node-set is empty.

Example

This op script displays some results of **math:highest**(), including the case where an empty node-set is returned because one of the nodes is not a valid number.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* node-set with a single maximum */
        var $rtf = {
            <number> 1;
            <number> 2;
            <number> 3;
        }
        var $ns = exsl:node-set( $rtf )/number;

        <output> "Highest of first node-set:";
        for-each( math:highest( $ns ) ) {
            <output> .;
        }

        /* node-set with multiple maximum */
        var $rtf2 = {
            <number> 100;
            <number> 200;
            <number> 200;
        }
        var $ns2 = exsl:node-set( $rtf2 )/number;

        <output> "Highest of second node-set:";
        for-each( math:highest( $ns2 ) ) {
            <output> .;
        }

        /* node-set with NaN */
        var $rtf3 = {
            <number> 150;
            <number> "string";
            <number> 30;
        }
        var $ns3 = exsl:node-set( $rtf3 )/number;

        <output> "Highest of third node-set:";
        for-each( math:highest( $ns3 ) ) {
            <output> .;
        }
    }
}
```

Output

```
Highest of first node-set:
3
Highest of second node-set:
200
200
Highest of third node-set:
```

math:log()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:log**(number)

Description

The **math:log**() function returns the natural logarithm (logarithm to the base e) of the number argument.

Example

This op script displays some results of **math:log**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:log( 5 );
        <output> math:log( 1.3 );
        <output> math:log( 100 );
    }
}
```

Output

```
1.6094379124341
0.262364264467491
4.605170185988092
```

math:lowest()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

node-set **math:lowest**(node-set *[numbers]*)

Description

The **math:lowest**() function returns the node in the node-set that contains the minimum numeric value. If multiple nodes share the same minimum value then they are all returned. If any nodes evaluate to NaN then the returned node-set is empty.

Example

This op script displays some results of **math:lowest**(), including the case where an empty node-set is returned because one of the nodes is not a valid number.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* node-set with a single minimum */
        var $rtf = {
            <number> 1;
            <number> 2;
            <number> -3;
        }
        var $ns = exsl:node-set( $rtf )/number;

        <output> "Lowest of first node-set:";
        for-each( math:lowest( $ns ) ) {
            <output> .;
        }

        /* node-set with multiple minimum */
        var $rtf2 = {
            <number> 100;
            <number> 100;
            <number> 200;
```

```
            }
            var $ns2 = exsl:node-set( $rtf2 )/number;

            <output> "Lowest of second node-set:";
            for-each( math:lowest( $ns2 ) ) {
                <output> .;
            }

            /* node-set with NaN */
            var $rtf3 = {
                <number> 150;
                <number> "string";
                <number> 30;
            }
            var $ns3 = exsl:node-set( $rtf3 )/number;

            <output> "Lowest of third node-set:";
            for-each( math:lowest( $ns3 ) ) {
                <output> .;
            }
        }
    }
```

Output

```
Lowest of first node-set:
-3
Lowest of second node-set:
100
100
Lowest of third node-set:
```

math:max()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:max**(node-set *[numbers]*)

Description

The **math:max**() function returns the highest number from among the numeric values of the nodes within the node-set argument. If any nodes evaluate to NaN then the result is NaN.

Example

This op script displays some results of **math:max**(), including the case where NaN is returned.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $rtf = {
            <number> 1;
            <number> 2;
            <number> 3;
        }
        var $ns = exsl:node-set( $rtf )/number;
        <output> "Highest value of first node-set: " _ math:max( $ns );

        var $rtf2 = {
            <number> -100;
            <number> -101;
        }
        var $ns2 = exsl:node-set( $rtf2 )/number;
        <output> "Highest value of second node-set: " _ math:max( $ns2 );

        /* node-set with NaN */
        var $rtf3 = {
            <number> 150;
            <number> "string";
            <number> 30;
        }
        var $ns3 = exsl:node-set( $rtf3 )/number;
        <output> "Highest value of third node-set: " _ math:max( $ns3 );

    }
}
```

Output

```
Highest value of first node-set: 3
Highest value of second node-set: -100
Highest value of third node-set: NaN
```

math:min()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:min**(node-set *[numbers]*)

Description

The **math:min**() function returns the lowest number from among the numeric values of the nodes within the node-set argument. If any nodes evaluate to NaN then the result is NaN.

Example

This op script displays some results of **math:min**(), including the case where NaN is returned.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $rtf = {
            <number> 1;
            <number> 2;
            <number> 3;
        }
        var $ns = exsl:node-set( $rtf )/number;
        <output> "Lowest value of first node-set: " _ math:min( $ns );

        var $rtf2 = {
            <number> -100;
            <number> -101;
        }
        var $ns2 = exsl:node-set( $rtf2 )/number;
        <output> "Lowest value of second node-set: " _ math:min( $ns2 );

        /* node-set with NaN */
        var $rtf3 = {
            <number> 150;
            <number> "string";
            <number> 30;
        }
        var $ns3 = exsl:node-set( $rtf3 )/number;
        <output> "Lowest value of third node-set: " _ math:min( $ns3 );

    }
}
```

Output

```
Lowest value of first node-set: 1
```

```
Lowest value of second node-set: -101
Lowest value of third node-set: NaN
```

math:power()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:power**(number, number *[power]*)

Description

The **math:power**() function returns the first number argument raised to the power of the second number argument.

Example

This op script displays some results of **math:power**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:power( 3, 0 );
        <output> math:power( 2, 1 );
        <output> math:power( 3, 2 );
        <output> math:power( 9, 3 );
    }
}
```

Output

```
1
2
9
729
```

math:random()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:random**()

Description

The **math:random**() function returns a random number with a minimum value of 0 and a maximum value of 1.

N*ote: Prior to Junos 10.2, the math:random() function was not seeded correctly, so it would return the same sequence of numbers each time a script was executed.*

Example

This op script displays some results of **math:random**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:random();
        <output> math:random();
        <output> math:random();
    }
}
```

Output

```
0.663341003779015
0.772250513905776
0.214387214376771
```

math:sin()

Source: EXSLT
Namespace: http://exslt.org/math

Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:sin**(number)

Description

The **math:sin**() function returns the sine radian value of the number argument.

Example

This op script computes the sine value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:sin( 1 );
        <output> math:sin( 0 );
        <output> math:sin( -0.5 );
    }
}
```

Output

```
0.841470984807897
0
-0.479425538604203
```

math:sqrt()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:sqrt**(number)

Description

The **math:sqrt**() function returns the square root of the number argument.

Example

This op script computes the square root of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:sqrt( 1 );
        <output> math:sqrt( 9 );
        <output> math:sqrt( 500 );
    }
}
```

Output

```
1
3
22.3606797749979
```

math:tan()

Source: EXSLT
Namespace: http://exslt.org/math
Common Prefix: math
Minimum Version: Junos 9.4

Syntax

number **math:tan**(number)

Description

The **math:tan**() function returns the tangent radian value of the number argument.

Example

This op script computes the tangent value of a few different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns math = "http://exslt.org/math";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> math:tan( 100 );
        <output> math:tan( 0 );
        <output> math:tan( -5000 );
    }
}
```

Output

```
-0.587213915156929
0
6.387642202844122
```

name()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **name**()

string **name**(node-set)

Description

Node names can consist of both a local part as well as a namespace. The **name**() function returns the qualified name of a node, which is the namespace prefix (if one exists) and the local name, separated by a colon. Example: <xnm:error> where xnm is the namespace prefix and error is the local name.

If the **name**() function is called with no argument then it returns the qualified name of the context node. Otherwise, it returns the qualified name of the node in the node-set argument. If multiple nodes are included within the node-set then the first node in document order is selected.

If the provided node has no name, or the node-set argument is empty, then an empty string is returned.

Example

This examples shows the qualified name of an element and its attributes being returned through the **name**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $results = jcs:invoke("get-system-uptime-information");
    var $date-time = $results/current-time/date-time;

    expr jcs:output( "Element name: ", name( $date-time ) );
    expr jcs:output( "Attribute name: ", name( $date-time/@* ) );
}
```

jcs:invoke() results:

```
<system-uptime-information>
    <current-time>
        <date-time junos:seconds="1262716723">2010-01-05 18:38:43 UTC</date-time>
    </current-time>
    <system-booted-time>
        <date-time junos:seconds="1258525183">2009-11-18 06:19:43 UTC</date-time>
        <time-length junos:seconds="4191540">6w6d 12:19</time-length>
    </system-booted-time>
    <protocols-started-time>
        <date-time junos:seconds="1262699988">2010-01-05 13:59:48 UTC</date-time>
        <time-length junos:seconds="16735">04:38:55</time-length>
    </protocols-started-time>
    <last-configured-time>
        <date-time junos:seconds="1262700008">2010-01-05 14:00:08 UTC</date-time>
        <time-length junos:seconds="16715">04:38:35</time-length>
        <user>jnpr</user>
    </last-configured-time>
    <uptime-information>
        <date-time junos:seconds="1262716723">6:38PM</date-time>
        <up-time junos:seconds="4191570">48 days, 12:19</up-time>
        <active-user-count junos:format="2 users">2</active-user-count>
        <load-average-1>0.12</load-average-1>
        <load-average-5>0.08</load-average-5>
        <load-average-15>0.08</load-average-15>
    </uptime-information>
</system-uptime-information>
```

Output

```
Element name: date-time
Attribute name: junos:seconds
```

namespace-uri()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **namespace-uri**()

string **namespace-uri**(node-set)

Description

Node names can consist of both a local part as well as a namespace. The **namespace-uri**() function returns the URI of the namespace (not its defined prefix). If called with no argument then it returns the namespace-uri of the context node. Otherwise, it returns the namespace-uri of the node in the node-set argument. If multiple nodes are included within the node-set then the first node in document order is selected.

If the provided node has no namespace-uri, or the node-set argument is empty, then an empty string is returned.

Example

In this op script example an incorrect Junos XML API request is performed so that a <xnm:error> is returned by **jcs:invoke**(). Both the qualified and name-space-uri of the <xnm:error> node are displayed by using the **name**() and **namespace-uri**() functions.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $results = jcs:invoke("invalid-rpc");
    expr jcs:output( "Qualified name: ", name( $results ) );
    expr jcs:output( "Namespace-uri: ", namespace-uri( $results ) );
}
        Output
Qualified name: xnm:error
Namespace-uri: http://xml.juniper.net/xnm/1.1/xnm
```

normalize-space()

Source: XPath

Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **normalize-space**()

string **normalize-space**(string)

Description

The **normalize-space**() function is used to remove excess whitespace from a string. It performs the following two modifications to the string:

- All leading and trailing whitespace is removed.
- All internal whitespace sequences are replaced with a single space.

Whitespace characters include space (“ ”), tab (“\t”), newline (“\n”), and carriage return (“\r”). If the argument provided is not a string then it is translated to a string before processing. If no argument is provided then the string value of the context node is used.

Example

This op script example shows the effect of the **normalize-space**() function on a string. All leading whitespace is removed and every whitespace character or sequence of characters within the string is replaced by a single space character.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $example-string = "\t \nOne\nTwo\nThree    Four\tFive";
        <output> "Original:" _ $example-string;
        <output> "Normalized:" _ normalize-space( $example-string );
    }
}
        Output
Original:
One
Two
Three    Four   Five
Normalized:One Two Three Four Five
```

not()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **not**(boolean)

Description

There is no not operator in SLAX. Instead, the **not**() function can be used to return the opposite of the provided boolean value. If the argument is true then **not**() returns false. If the argument is false then **not**() returns true.

Example

This op script example shows how to use the **not**() function to return the opposite of the original boolean value. In the case below, the configuration does not have telnet enabled and the script results reflect that.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $configuration-rpc = {
        <get-configuration database="committed" inherit="inherit">;
    }
    var $configuration = jcs:invoke( $configuration-rpc );

    if( not( $configuration/system/services/telnet ) ) {
        expr jcs:output( "The telnet protocol is not enabled" );
    }
}
```

System services configuration

```
services {
    ssh;
}
```

Output

```
The telnet protocol is not enabled
```

number()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **number**()

number **number**(object)

Description

The **number**() function returns the number value of the provided argument. If no argument is included then it defaults to a node-set that contains the context node. Data types are converted to number values in the following ways:

- String – Strings that contain only valid numeric content are converted to their equivalent number, otherwise they are converted to NaN. For a string to consist of valid numeric content, spaces can only be present prior to the start of the number, no commas are allowed, and periods must be used as the decimal point if a decimal is present. The following scientific notation forms are also valid: 1.23e5, 1.23e+5, 0.12e-3.
- Boolean – true is converted to 1, false to 0.
- Node-set –Converted to a string in the same way as the string() function, and the string result is then converted to a number.
- Result tree fragment – Converted to a string in the same way as the string() function, and the string result is then converted to a number.

Example

The following op script includes various examples of the **number**() function returning the number value of different data types.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $rtf = {
            <num> 1;
            <num> 23;
            <num> 14;
        }
```

```
        var $node-set := {
            copy-of $rtf;
        }
        var $string1 = "123.45";
        var $string2 = "-50";
        var $boolean = false();

        <output> number( $rtf );
        <output> number( $node-set/num[2] );
        <output> number( $string1 );
        <output> number( $string2 );
        <output> number( $boolean );
        <output> number( "string" );
    }
}
```

Output

```
12314
23
123.45
-50
0
NaN
```

position()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **position**()

Description

The **position**() function returns the current context position. The context position is the index of the node within the node-set being evaluated by a predicate, or if **position**() is being used outside of a predicate then it is the index of the current node within the current node list. The initial position is 1 and the final position is equal to the context size, which can be retrieved through the **last**() function.

Example

This op script example shows the effect of using the **position**() function in both location path predicates as well as within for-each loops.

Code

```
version 1.0;
```

```
    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match / {
        var $host-name-set := {
            <host-name> "PE1";
            <host-name> "P1";
            <host-name> "P2";
            <host-name> "PE2";
        }

        var $first-host-name = $host-name-set/host-name[ position() == 1 ];
        expr jcs:output( "First host-name: ", $first-host-name );

        var $first-p-host-name = $host-name-set/host-name[not(starts-with(.,"PE"))][position() ==
1];
        expr jcs:output( "First P host-name: ", $first-p-host-name );

        expr jcs:output( "All host-names:" );
        for-each( $host-name-set/host-name ) {
            expr jcs:output( position(), ": ", . );
        }

        expr jcs:output( "P host-names only:" );
        for-each( $host-name-set/host-name[ not(starts-with( ., "PE" ))] ) {
            expr jcs:output( position(), ": ", . );
        }
    }
```

Output

```
First host-name: PE1
First P host-name: P1
All host-names:
1: PE1
2: P1
3: P2
4: PE2
P host-names only:
1: P1
2: P2
```

round()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **round**(number)

Description

The **round**() function is one of the available SLAX rounding functions. It rounds the number argument to the closest integer. If there are two integers of equivalent distance from the argument then the larger integer is returned. Argument values of NaN return NaN, positive infinity return positive infinity, and negative infinity return negative infinity.

Example

This op script example shows the result of the **round**() function with many different numbers.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> round( 1 );
        <output> round( 0.9 );
        <output> round( 5.1 );
        <output> round( 5.5 );
        <output> round( 5.999999999 );
        <output> round( -3.9 );
        <output> round( -0.9 );
        <output> round( -3.1 );
    }
}
```

Output

```
1
1
5
6
6
-4
-1
-3
```

saxon:eval()

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 9.4

Syntax

object *[result]* **saxon:eval**(external-object *[stored expression]*)

Description

The **saxon:eval**() function dynamically processes an XPath expression, which is provided as a stored expression external object. The stored expression is evaluated as if its original string value were written in the code in place of the **saxon:eval**() function call. This means that the context information is the same as the location where the **saxon:eval**() function was called, including:

- Context node, size, and position
- Current node
- Variables and parameters
- Functions
- Namespace declarations
- Keys
- Decimal formats

The **saxon:eval**() function is always paired with the **saxon:expression**() function, because only the latter function is capable of generating stored-expressions. After the XPath expression is evaluated by **saxon:eval**(), it returns the result of the expression, which could be in any of the standard data types.

Potential uses

- Calling a function that is determined dynamically, such as through user input.
- Calling a function with a dynamic number of arguments.
- Performing an operation or comparison where the operator is determined dynamically.
- Building location paths dynamically, such as through user input.

Note: Converting a string into a stored expression through the **saxon:expression**() function and then processing the expression with the **saxon:eval**() function gives equivalent behavior to providing the string to the **saxon:evaluate**() function and processing the XPath expression in a single step.

Example

This op script demonstrates how **saxon:expression**() can be used in combination with **saxon:eval**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns saxon = "http://icl.com/saxon";

import "../import/junos.xsl";

match / {
```

```
    <op-script-results> {

        var $parameter = jcs:get-input("Display which parameter? (Don't include $): ");

        /* Build the stored expression */
        var $expression = saxon:expression(concat( "$", $parameter));

        /* Execute the stored expression */
        expr jcs:output( $parameter, " value is ", saxon:eval( $expression ) );
    }
}
```

Output

```
Display which parameter? (Don't include $): localtime-iso
localtime-iso value is 2010-11-09 21:57:15 UTC
```

saxon:evaluate()

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 9.4

Syntax

object *[result]* **saxon:evaluate**(string *[XPath]*)

Description

The **saxon:evaluate**() function processes an XPath expression dynamically. It takes a single string argument, which is the expression that should be executed. This expression is evaluated as if the exact string were written in the code in place of the **saxon:evaluate**() function call. This means that the context information is the same as the location where the **saxon:evaluate**() function was called, including:

- Context node, size, and position
- Current node
- Variables and parameters
- Functions
- Namespace declarations
- Keys
- Decimal formats

If the string argument is blank or an invalid XPath expression then the function will fail with an error; otherwise, the function returns the result of the expression, which could be in any of the standard data types.

When using this function, keep the following in mind:

- These SLAX-specific operators are not supported in the expression:

- "==" equality operator
  - Must use "="
- "&&" and operator
  - Must use "and"
- "||" or operator
  - Must use "or"
- "_" string concatenating operator
  - Must use **concat**() function

- Functions can be called but templates cannot.
- Do not terminate the expression with a semicolon.
- Do not start the expression with the **expr** statement.
- The namespace of any functions used within the expression must be defined within the SLAX script.

Potential uses

- Calling a function that is determined dynamically, such as through user input.
- Calling a function with a dynamic number of arguments.
- Performing an operation or comparison where the operator is determined dynamically.
- Building location paths dynamically, such as through user input.

Note: The **dyn:evaluate**() and **saxon:evaluate**() functions have equivalent functionality.

Example

This op script demonstrates two example uses of **saxon:evaluate**(). The first example needs to use **saxon:evaluate**() because the function to be called is determined dynamically. The second example actually didn't need to use **saxon:evaluate**(), but it was included to point out that the equality operator needs to be "=" rather than "==". Also, notice that the necessary namespaces to run the expected functions have been defined in the script.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns saxon = "http://icl.com/saxon";
ns math = "http://exslt.org/math";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Ask for the function */
        var $prompt = "Choose an EXSLT math function\n" _
```

```
                    "(abs,acos,asin,atan,cos,exp,log,sin,sqrt,tan)\n" _
                    "Enter selection: ";
            var $function = jcs:get-input( $prompt );

            /* Ask for the number */
            var $number = jcs:get-input( "Enter number to compute: " );

            /* Retrieve the result */
            var $expression = "math:" _ $function _ "( $number )";
            expr jcs:output( "Expression used: ", $expression );
            expr jcs:output(  "The answer is ", saxon:evaluate( $expression ) );

            /* This example allows the user to select an index to tokenize */
            var $example := {
                <node> "one";
                <node> "two";
                <node> "three";
                <node> "four";
            }
            expr jcs:output( "_\r \nOptions:" );
            for-each( $example/node ) {
                expr jcs:output( . );
            }
            var $index = jcs:get-input( "Select index to tokenize (1-4): " );

            /* dyn:evaluate must use XSLT syntax, so equality operator is = instead of == */
            var $expression2 = "str:tokenize( $example/node[ position() = " _ $index _ "], \"\"
)";

            /* retrieve and display the tokenized string */
            var $tokens = saxon:evaluate( $expression2 );
            for-each( $tokens ) {
                expr jcs:output( . );
            }
        }
    }
```

Output

```
Choose an EXSLT math function
(abs,acos,asin,atan,cos,exp,log,sin,sqrt,tan)
Enter selection: log
Enter number to compute: 3
Expression used: math:log( $number )
The answer is 1.09861228866811
Options:
one
two
three
four
Select index to tokenize (1-4): 1
o
n
e
```

saxon:expression()

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 9.4

Syntax

external-object *[stored expression]* **saxon:expression**(string *[XPath]*)

Description

The **saxon:expression**() function converts a XPath string into a stored expression external object, which is a special data type that contains a XPath expression. This stored expression object can only be used by the **saxon:eval**() function. It cannot be converted into any other data type or used in any other way.

If the string argument is blank, or an invalid XPath expression, then the function will fail with an error.

When using this function, keep the following in mind:

- These SLAX-specific operators are not supported in the expression:
  - "==" equality operator
    - Must use "="
  - "&&" and operator
    - Must use "and"
  - "||" or operator
    - Must use "or"
  - "_" string concatenating operator
    - Must use **concat**() function
- Functions can be called but templates cannot.
- Do not terminate the expression with a semicolon.
- Do not start the expression with the **expr** statement.
- The namespace of any functions used within the expression must be defined within the SLAX script.

Potential uses

- Calling a function that is determined dynamically, such as through user input.
- Calling a function with a dynamic number of arguments.
- Performing an operation or comparison where the operator is determined dynamically.
- Building location paths dynamically, such as through user input.

Note: Converting a string into a stored expression through the **saxon:expression**() function and then processing the expression with the **saxon:eval**() function gives equivalent behavior to providing the string to the **saxon:evaluate**() function and processing the XPath expression in a single step.

Example

This op script demonstrates how **saxon:expression**() can be used in combination with **saxon:eval**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns saxon = "http://icl.com/saxon";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $parameter = jcs:get-input("Display which parameter? (Don't include $): ");

        /* Build the stored expression */
        var $expression = saxon:expression(concat( "$", $parameter));

        /* Execute the stored expression */
        expr jcs:output( $parameter, " value is ", saxon:eval( $expression ) );
    }
}
```

Output

```
Display which parameter? (Don't include $): localtime-iso
localtime-iso value is 2010-11-09 21:57:15 UTC
```

saxon:line-number()

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 9.4

Syntax

number *[line]* **saxon:line-number**()

number *[line]* **saxon:line-number**(node-set)

Description

The **saxon:line-number**() function returns the line number of a node within its XML document. If no argument is provided then the context node's line number is returned. If a node-set is provided then the line number of the node that is first in document order is returned. If the node-set argument is empty, or if the selected node is a root node, then -1 is returned.

If the XML document was read through the **document**() function then the returned line number represents the line number of the file where the element is located, starting with 1.

If the XML document was created through a **jcs:invoke**() or **jcs:execute**() function call then the line number of the returned node starts with 7.

Otherwise, most other functions do not support line numbers and return 0. This also includes node-sets that were converted from result tree fragments.

If the XML document is the source tree of an event script then <event-script-input> has a line number of 1.

If the XML document is the source tree of a commit script then <commit-script-input> has a line number of 2, <configuration> has a line number of 4, and <version> has a line number of 5, followed by the configuration contents.

Example

This op script displays the line numbers of the nodes in a XML document.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns saxon = "http://icl.com/saxon";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Read file */
        var $file = document( "/var/tmp/example.xml" );

        /* Display all the line-numbers */
        for-each( $file//* ) {
            <output> name() _ " = " _ saxon:line-number();
        }
    }
}
```

example.xml

```
<?xml version="1.0"?>
<users>
    <user>
        <name>jnpr</name>
    </user>
</users>
```

Output

```
users = 2
user = 3
name = 4
```

saxon:node-set()

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 8.2

Syntax

node-set **saxon:node-set**(node-set)

node-set **saxon:node-set**(result-tree-fragment)

Description

The **saxon:node-set**() function is used to convert a result tree fragment into a node-set. The behavior differs depending on the data type of the argument:

- Result tree fragment - A new XML document is created consisting of the nodes expressed by the result tree fragment. The returned node-set contains the root node of the new XML document.
- Node-set – The node-set argument is returned as the result.

One common stumbling block when using the **saxon:node-set**() function is remembering that a converted result tree fragment results in a node-set that contains the root node of the new XML document, rather than all of the XML children. For example, the following result tree fragment:

```
var $rtf = {
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/1";
}
```

would be converted into a node-set with a single node: the root-node of the XML document, of which both of the <interface> nodes would be children; however, if a script writer wishes to have the node-set contain all the top-level child nodes, rather than the root node, then the following code can be used:

```
var $ns = saxon:node-set( $rtf )/*;
```

The above example first converts the result tree fragment variable into a node-set, the returned root node is then subjected to a location-path to retrieve its children, and it is these child nodes that are assigned as a node-set to the $ns variable.

Note: The libxslt:node-set(), saxon:node-set(), and xt:node-set() functions are all identical. They also provide similar functionality as the exsl:node-set() function, except that the latter function is capable of converting strings, booleans, and numbers into node-sets as well.

Example

This op script demonstrates how the various node-set functions can be used both to convert a result tree fragment variable into a node-set as well as a string into a node-set.

Code

```
version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns libxslt = "http://xmlsoft.org/XSLT/namespace";
ns saxon = "http://icl.com/saxon";
ns xt = "http://www.jclark.com/xt";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";


match / {
    <op-script-results> {
        var $rtf = {
            <node> "one";
            <node> "two";
            <node> "three";
        }
        /* Convert RTF into node-set using all four methods */
        var $l-ns = libxslt:node-set( $rtf );
        var $s-ns = saxon:node-set( $rtf );
        var $x-ns = xt:node-set( $rtf );
        var $e-ns = exsl:node-set( $rtf );

        /* Display counts */
        <output> "libxslt #" _ count( $l-ns/node );
        <output> "saxon #" _ count( $s-ns/node );
        <output> "xt #" _ count( $x-ns/node );
        <output> "exsl #" _ count( $e-ns/node );

        /* Only exsl:node-set can convert string, booleans, and numbers */
        for-each( exsl:node-set( "Example text" ) ) {
            <output> "String converted by exsl:node-set() into text node: " _ .;
        }
    }
}
```

Output

```
libxslt #3
saxon #3
xt #3
exsl #3
String converted by exsl:node-set() into text node: Example text
```

set:difference()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

node-set **set:difference**(node-set *[target]*, node-set)

Description

The **set:difference**() function returns a node-set that contains all the nodes from the first node-set argument that are not present in the second node-set argument.

It is the nodes themselves, rather than their values, that are compared. In other words, multiple nodes containing the same value are still considered different nodes, even though their values are identical.

Example

This op script demonstrates how the **set:difference**() function could be used. The script assembles two separate node-sets based on the same route data and then compares the two using the **set:difference**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get route information */
        var $routes = jcs:invoke( "get-route-information" );

        /* static routes */
        var $static-routes = $routes//rt-entry[protocol-name == "Static"];

        /* active routes */
        var $active-routes = $routes//rt-entry[current-active];

        /* Which routes are static but are not active? */
        var $difference = set:difference( $static-routes, $active-routes );

        for-each( $difference ) {
            <output> ../rt-destination _ " nh " _ nh/to;
        }
    }
}
```

jcs:invoke() results

```
<route-information>
    <!-- keepalive -->
    <route-table>
        <table-name>inet.0</table-name>
        <destination-count>4</destination-count>
        <total-route-count>5</total-route-count>
        <active-route-count>4</active-route-count>
        <holddown-route-count>0</holddown-route-count>
        <hidden-route-count>0</hidden-route-count>
        <rt junos:style="brief">
            <rt-destination>0.0.0.0/0</rt-destination>
            <rt-entry>
```

```
            <active-tag>*</active-tag>
            <current-active/>
            <last-active/>
            <protocol-name>Static</protocol-name>
            <preference>5</preference>
            <age junos:seconds="30771">08:32:51</age>
            <nh>
                <selected-next-hop/>
                <to>10.0.0.1</to>
                <via>ge-0/0/0.0</via>
            </nh>
        </rt-entry>
        <rt-entry>
            <active-tag> </active-tag>
            <protocol-name>Static</protocol-name>
            <preference>200</preference>
            <age junos:seconds="30823">08:33:43</age>
            <nh>
                <selected-next-hop/>
                <to>10.0.0.2</to>
                <via>ge-0/0/0.0</via>
            </nh>
        </rt-entry>
    </rt>
    <rt junos:style="brief">
        <rt-destination>10.0.0.0/24</rt-destination>
        <rt-entry>
            <active-tag>*</active-tag>
            <current-active/>
            <last-active/>
            <protocol-name>Direct</protocol-name>
            <preference>0</preference>
            <age junos:seconds="30823">08:33:43</age>
            <nh>
                <selected-next-hop/>
                <via>ge-0/0/0.0</via>
            </nh>
        </rt-entry>
    </rt>
    <rt junos:style="brief">
        <rt-destination>10.0.0.10/32</rt-destination>
        <rt-entry>
            <active-tag>*</active-tag>
            <current-active/>
            <last-active/>
            <protocol-name>Local</protocol-name>
            <preference>0</preference>
            <age junos:seconds="30831">08:33:51</age>
            <nh-type>Local</nh-type>
            <nh>
                <nh-local-interface>ge-0/0/0.0</nh-local-interface>
            </nh>
        </rt-entry>
    </rt>
    <rt junos:style="brief">
        <rt-destination>192.168.255.254/32</rt-destination>
        <rt-entry>
            <active-tag>*</active-tag>
            <current-active/>
            <last-active/>
            <protocol-name>Static</protocol-name>
            <preference>5</preference>
            <age junos:seconds="30823">08:33:43</age>
```

```
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.2</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
    </route-table>
</route-information>
```

Output

```
0.0.0.0/0 nh 10.0.0.2
```

set:distinct()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

node-set **set:distinct**(node-set)

Description

The **set:distinct**() function returns all the nodes from the node-set argument that have a unique string value. If multiple nodes share the same string value then the first node in document order is returned and the others that have that string value are not included.

Example

This op script demonstrates how the **set:distinct**() function could be used to retrieve the names of all the protocols from which routes in the routing table are learned.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get route information */
```

```
        var $routes = jcs:invoke( "get-route-information" );

        /* Collect all the protocol names */
        var $protocols = $routes//protocol-name;

        /* Get distinct names */
        var $names = set:distinct( $protocols );

        for-each( $names ) {
            /* alphabetical */
            <xsl:sort>;
            <output> .;
        }
    }
}
```

jcs:invoke() results

```
<route-information>
    <!-- keepalive -->
    <route-table>
        <table-name>inet.0</table-name>
        <destination-count>4</destination-count>
        <total-route-count>4</total-route-count>
        <active-route-count>4</active-route-count>
        <holddown-route-count>0</holddown-route-count>
        <hidden-route-count>0</hidden-route-count>
        <rt junos:style="brief">
            <rt-destination>0.0.0.0/0</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="31646">08:47:26</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.1</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.0/24</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Direct</protocol-name>
                <preference>0</preference>
                <age junos:seconds="31698">08:48:18</age>
                <nh>
                    <selected-next-hop/>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.10/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
```

```
                <current-active/>
                <last-active/>
                <protocol-name>Local</protocol-name>
                <preference>0</preference>
                <age junos:seconds="31706">08:48:26</age>
                <nh-type>Local</nh-type>
                <nh>
                    <nh-local-interface>ge-0/0/0.0</nh-local-interface>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>192.168.0.0/16</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="465">00:07:45</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.1</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
    </route-table>
</route-information>
```

Output

```
Direct
Local
Static
```

set:has-same-node()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

boolean **set:has-same-node**(node-set, node-set)

Description

The **set:has-same-node**() function returns true if the two node-set arguments share at least one node in common. If there are no nodes that are shared between both node-sets then it returns false.

It is the nodes themselves, rather than their values, that are compared. In other words, multiple nodes containing the same value are still considered different nodes even though their values are identical.

Example

This op script demonstrates the usage of **set:has-same-node()**.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get interface information */
        var $rpc = {
            <get-interface-information> {
                <terse>;
                <interface-name> "fe*";
            }
        }

        /* Get logical interfaces */
        var $interfaces = jcs:invoke( $rpc )/physical-interface/logical-interface;

        /* IPv4 logical interfaces */
        var $ipv4-interfaces = $interfaces[address-family[address-family-name == "inet"]];

        /* ISO logical interfaces */
        var $iso-interfaces = $interfaces[address-family[address-family-name == "iso"]];

        /* Display IPv4 count */
        <output> "There are " _ count( $ipv4-interfaces ) _ " interfaces with IPv4 enabled.";

        /* Display ISO count */
        <output> "There are " _ count( $iso-interfaces ) _ " interfaces with ISO enabled.";

        /* Is there at least one with both? */
        var $boolean-value = set:has-same-node( $ipv4-interfaces, $iso-interfaces );
        <output> "One or more interfaces has both = " _ $boolean-value;

    }
}
```

jcs:invoke() results

```
<interface-information junos:style="terse">
    <physical-interface>
        <name>fe-0/0/2</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <logical-interface>
            <name>fe-0/0/2.0</name>
            <admin-status>up</admin-status>
            <oper-status>down</oper-status>
            <filter-information>
            </filter-information>
```

```
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.1.1/24</ifa-local>
            </interface-address>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/3</name>
    <admin-status>up</admin-status>
    <oper-status>up</oper-status>
    <logical-interface>
        <name>fe-0/0/3.0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.2.1/24</ifa-local>
            </interface-address>
        </address-family>
        <address-family>
            <address-family-name>iso</address-family-name>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/4</name>
    <admin-status>up</admin-status>
    <oper-status>up</oper-status>
    <description>Other</description>
    <logical-interface>
        <name>fe-0/0/4.0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.3.1/24</ifa-local>
            </interface-address>
        </address-family>
        <address-family>
            <address-family-name>iso</address-family-name>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/5</name>
    <admin-status>up</admin-status>
    <oper-status>down</oper-status>
    <logical-interface>
        <name>fe-0/0/5.0</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
```

```
                <interface-address>
                    <ifa-local junos:emit="emit">192.168.4.1/24</ifa-local>
                </interface-address>
            </address-family>
            <address-family>
                <address-family-name>iso</address-family-name>
            </address-family>
        </logical-interface>
    </physical-interface>
    <physical-interface>
        <name>fe-0/0/6</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
    </physical-interface>
    <physical-interface>
        <name>fe-0/0/7</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
    </physical-interface>
</interface-information>
```

Output

```
There are 4 interfaces with IPv4 enabled.
There are 3 interfaces with ISO enabled.
One or more interfaces has both = true
```

set:intersection()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

node-set **set:intersection**(node-set, node-set)

Description

The **set:intersection**() function returns a node-set that contains all the nodes that are present within both of the node-set arguments.

It is the nodes themselves, rather than their values, that are compared. In other words, multiple nodes containing the same value are still considered different nodes even though their values are identical.

Example

This op script demonstrates how **set:intersection**() can be used to display a list of nodes that are present in both of two different node-sets.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Get interface information */
        var $rpc = {
            <get-interface-information> {
                <terse>;
                <interface-name> "fe*";
            }
        }

        /* Get logical interfaces */
        var $interfaces = jcs:invoke( $rpc )/physical-interface/logical-interface;

        /* IPv4 logical interfaces */
        var $ipv4-interfaces = $interfaces[address-family[address-family-name == "inet"]];

        /* ISO logical interfaces */
        var $iso-interfaces = $interfaces[address-family[address-family-name == "iso"]];

        /* Display IPv4 interfaces */
        <output> "Interfaces with IPv4 enabled:";
        for-each( $ipv4-interfaces ) {
            <output> name;
        }

        /* Display ISO interfaces */
        <output> "Interfaces with ISO enabled:";
        for-each( $iso-interfaces ) {
            <output> name;
        }

        /* Display both */
        <output> "Interfaces with both enabled:";
        for-each( set:intersection( $ipv4-interfaces, $iso-interfaces ) ) {
            <output> name;
        }
    }
}
```

jcs:invoke() results

```
<interface-information junos:style="terse">
    <physical-interface>
        <name>fe-0/0/2</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <logical-interface>
            <name>fe-0/0/2.0</name>
            <admin-status>up</admin-status>
            <oper-status>down</oper-status>
```

```
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.1.1/24</ifa-local>
            </interface-address>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/3</name>
    <admin-status>up</admin-status>
    <oper-status>up</oper-status>
    <logical-interface>
        <name>fe-0/0/3.0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.2.1/24</ifa-local>
            </interface-address>
        </address-family>
        <address-family>
            <address-family-name>iso</address-family-name>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/4</name>
    <admin-status>up</admin-status>
    <oper-status>up</oper-status>
    <description>Other</description>
    <logical-interface>
        <name>fe-0/0/4.0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <filter-information>
        </filter-information>
        <address-family>
            <address-family-name>inet</address-family-name>
            <interface-address>
                <ifa-local junos:emit="emit">192.168.3.1/24</ifa-local>
            </interface-address>
        </address-family>
        <address-family>
            <address-family-name>iso</address-family-name>
        </address-family>
    </logical-interface>
</physical-interface>
<physical-interface>
    <name>fe-0/0/5</name>
    <admin-status>up</admin-status>
    <oper-status>down</oper-status>
    <logical-interface>
        <name>fe-0/0/5.0</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <filter-information>
        </filter-information>
```

```
            <address-family>
                <address-family-name>inet</address-family-name>
                <interface-address>
                    <ifa-local junos:emit="emit">192.168.4.1/24</ifa-local>
                </interface-address>
            </address-family>
            <address-family>
                <address-family-name>iso</address-family-name>
            </address-family>
        </logical-interface>
    </physical-interface>
    <physical-interface>
        <name>fe-0/0/6</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
    </physical-interface>
    <physical-interface>
        <name>fe-0/0/7</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
    </physical-interface>
</interface-information>
```

Output

```
Interfaces with IPv4 enabled:
fe-0/0/2.0
fe-0/0/3.0
fe-0/0/4.0
fe-0/0/5.0
Interfaces with ISO enabled:
fe-0/0/3.0
fe-0/0/4.0
fe-0/0/5.0
Interfaces with both enabled:
fe-0/0/3.0
fe-0/0/4.0
fe-0/0/5.0
```

set:leading()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

node-set **set:leading**(node-set *[target]*, node-set *[node]*)

Description

The **set:leading**() function takes two node-set arguments and returns all the nodes of the first node-set that precede, in document order, the first node of the second node-set argument. If the second node-set argument is

empty then the full first node-set argument is returned. If the first node of the second node-set argument is not found in the first node-set argument then an empty node-set is returned.

It is the nodes themselves, rather than their values, that are compared. In other words, multiple nodes containing the same value are still considered different nodes even though their values are identical.

Example

This op script demonstrates how **set:leading**() could be used to return all the syslog lines that precede a particular log line.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve output of syslog file */
        var $rpc = {
            <file-show> {
                <filename> "/var/log/syslog";
            }
        }
        var $file-content = jcs:invoke( $rpc );

        /* Break into lines */
        var $lines = jcs:break-lines( $file-content )[ string-length() > 0 ];

        /* Find the RPD_EXIT line */
        var $rpd-exit-line = $lines[ contains( ., "RPD_EXIT" ) ];

        /* Display the lines that appear prior to RPD exit line */
        <output> "RPD_EXIT log:\n" _ $rpd-exit-line;

        <output> "Preceding logs:";
        for-each( set:leading( $lines, $rpd-exit-line ) ) {
            <output> .;
        }
    }
}
```

jcs:invoke() results

```
    <file-content filename="/var/log/syslog" junos:seconds="1284959156" filesize="922"
encoding="text">
        Sep 20 04:49:29 srx210 clear-log[2043]: logfile cleared
        Sep 20 04:49:31  srx210 mgd[1154]: UI_DBASE_LOGIN_EVENT: User 'jnpr' entering
configuration mode
        Sep 20 04:49:32  srx210 mgd[1154]: UI_COMMIT: User 'jnpr' requested 'commit' operation
(comment: none)
        Sep 20 04:49:40  srx210 mgd[1154]: UI_DBASE_LOGOUT_EVENT: User 'jnpr' exiting
configuration mode
```

```
        Sep 20 04:49:51  srx210 su: jnpr to root on /dev/ttyp0
        Sep 20 04:49:53  srx210 rpd[2037]: RPD_SIGNAL_TERMINATE: first termination signal received
        Sep 20 04:49:53  srx210 rpd[2037]: RPD_EXIT: Exit rpd[2037] version 10.3R1.9 built by
builder on 2010-08-13 12:57:35 UTC, caller 587a18
        Sep 20 04:49:53  srx210 rpd[2064]: L2CKT acquiring mastership for primary
        Sep 20 04:49:53  srx210 rpd[2064]: L2VPN acquiring mastership for primary
        Sep 20 04:49:54  srx210 rpd[2064]: RPD_TASK_BEGIN: Commencing routing updates, version
10.3R1.9, built 2010-08-13 12:57:35 UTC by builder
    </file-content>
```

Output

```
    RPD_EXIT log:
    Sep 20 04:49:53  srx210 rpd[2037]: RPD_EXIT: Exit rpd[2037] version 10.3R1.9 built by builder
on 2010-08-13 12:57:35 UTC, caller 587a18
    Preceding logs:
    Sep 20 04:49:29 srx210 clear-log[2043]: logfile cleared
    Sep 20 04:49:31  srx210 mgd[1154]: UI_DBASE_LOGIN_EVENT: User 'jnpr' entering configuration
mode
    Sep 20 04:49:32  srx210 mgd[1154]: UI_COMMIT: User 'jnpr' requested 'commit' operation
(comment: none)
    Sep 20 04:49:40  srx210 mgd[1154]: UI_DBASE_LOGOUT_EVENT: User 'jnpr' exiting configuration
mode
    Sep 20 04:49:51  srx210 su: jnpr to root on /dev/ttyp0
    Sep 20 04:49:53  srx210 rpd[2037]: RPD_SIGNAL_TERMINATE: first termination signal received
```

set:trailing()

Source: EXSLT
Namespace: http://exslt.org/sets
Common Prefix: set
Minimum Version: Junos 9.4

Syntax

node-set **set:trailing**(node-set *[target]*, node-set *[node]*)

Description

The **set:trailing**() function takes two node-set arguments and returns all the nodes of the first node-set that follow, in document order, the first node of the second node-set argument. If the second node-set argument is empty then the full first node-set argument is returned. If the first node of the second node-set argument is not found in the first node-set argument then an empty node-set is returned.

It is the nodes themselves, rather than their values, that are compared. In other words, multiple nodes containing the same value are still considered different nodes, even though their values are identical.

Example

This op script demonstrates how **set:trailing**() could be used to return all the syslog lines that follow a particular log line.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns set = "http://exslt.org/sets";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Retrieve output of syslog file */
        var $rpc = {
            <file-show> {
                <filename> "/var/log/syslog";
            }
        }
        var $file-content = jcs:invoke( $rpc );

        /* Break into lines */
        var $lines = jcs:break-lines( $file-content )[ string-length() > 0 ];

        /* Find the RPD_EXIT line */
        var $rpd-exit-line = $lines[ contains( ., "RPD_EXIT" ) ];

        /* Display the lines that appear after RPD exit line */
        <output> "RPD_EXIT log:\n" _ $rpd-exit-line;

        <output> "Following logs:";
        for-each( set:trailing( $lines, $rpd-exit-line ) ) {
            <output> .;
        }
    }
}
```

jcs:invoke() results

```
    <file-content filename="/var/log/syslog" junos:seconds="1284960428" filesize="1375"
encoding="text">
        Sep 20 05:23:59 srx210 clear-log[2143]: logfile cleared
        Sep 20 05:24:14  srx210 mgd[2140]: UI_RESTART_EVENT: User 'jnpr' restarting daemon
'Routing protocols process'
        Sep 20 05:24:14  srx210 rpd[2064]: RPD_SIGNAL_TERMINATE: first termination signal
received
        Sep 20 05:24:14  srx210 rpd[2064]: RPD_EXIT: Exit rpd[2064] version 10.3R1.9 built by
builder on 2010-08-13 12:57:35 UTC, caller 587a18
        Sep 20 05:24:15  srx210 init: can not access /usr/sbin/jdiameterd: No such file or
directory
        Sep 20 05:24:15  srx210 init: can not access /usr/sbin/ipmid: No such file or directory
        Sep 20 05:24:15  srx210 rpd[2146]: L2CKT acquiring mastership for primary
        Sep 20 05:24:15  srx210 rpd[2146]: L2VPN acquiring mastership for primary
        Sep 20 05:24:16  srx210 rpd[2146]: RPD_TASK_BEGIN: Commencing routing updates, version
10.3R1.9, built 2010-08-13 12:57:35 UTC by builder
        Sep 20 05:24:20  srx210 mgd[2140]: UI_DBASE_LOGIN_EVENT: User 'jnpr' entering
configuration mode
        Sep 20 05:24:20  srx210 mgd[2140]: UI_COMMIT: User 'jnpr' requested 'commit' operation
(comment: none)
        Sep 20 05:24:30  srx210 mgd[2140]: UI_DBASE_LOGOUT_EVENT: User 'jnpr' exiting
configuration mode
```

```
        Sep 20 05:26:26  srx210 mgd[2140]: LIBJNX_EXEC_WEXIT: Command exited: PID 2163, status 1,
command '/bin/cp'
        Sep 20 05:26:28  srx210 mgd[2140]: LIBJNX_EXEC_WEXIT: Command exited: PID 2164, status 1,
command '/bin/cp'
    </file-content>
```

Output

```
    RPD_EXIT log:
    Sep 20 05:24:14  srx210 rpd[2064]: RPD_EXIT: Exit rpd[2064] version 10.3R1.9 built by builder
on 2010-08-13 12:57:35 UTC, caller 587a18
    Following logs:
    Sep 20 05:24:15  srx210 init: can not access /usr/sbin/jdiameterd: No such file or directory
    Sep 20 05:24:15  srx210 init: can not access /usr/sbin/ipmid: No such file or directory
    Sep 20 05:24:15  srx210 rpd[2146]: L2CKT acquiring mastership for primary
    Sep 20 05:24:15  srx210 rpd[2146]: L2VPN acquiring mastership for primary
    Sep 20 05:24:16  srx210 rpd[2146]: RPD_TASK_BEGIN: Commencing routing updates, version
10.3R1.9, built 2010-08-13 12:57:35 UTC by builder
    Sep 20 05:24:20  srx210 mgd[2140]: UI_DBASE_LOGIN_EVENT: User 'jnpr' entering configuration
mode
    Sep 20 05:24:20  srx210 mgd[2140]: UI_COMMIT: User 'jnpr' requested 'commit' operation
(comment: none)
    Sep 20 05:24:30  srx210 mgd[2140]: UI_DBASE_LOGOUT_EVENT: User 'jnpr' exiting configuration
mode
    Sep 20 05:26:26  srx210 mgd[2140]: LIBJNX_EXEC_WEXIT: Command exited: PID 2163, status 1,
command '/bin/cp'
    Sep 20 05:26:28  srx210 mgd[2140]: LIBJNX_EXEC_WEXIT: Command exited: PID 2164, status 1,
command '/bin/cp'
```

starts-with()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **starts-with**(string *[target]*, string *[substring]*)

Description

The **starts-with**() function is test if a string starts with a second string. The function returns true if the first string argument starts with the second string argument. Otherwise, it returns false.

Example

This example shows how to use the **starts-with**() function with a predicate to only return nodes with a string-value that starts with the desired value.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $rpc = {
        <get-interface-information> {
            <descriptions>;
        }
    }

    var $results = jcs:invoke( $rpc );

    /* Only process ge-* interfaces */
    for-each( $results/physical-interface[ starts-with( name, "ge-" ) ] ) {
        expr jcs:output("Name: ", name, " Description: ", description );
    }
}
```

jcs:invoke() results:

```
<interface-information junos:style="description">
    <physical-interface>
        <name>ge-0/0/0</name>
        <admin-status>up</admin-status>
        <oper-status>up</oper-status>
        <description>Management Interface</description>
    </physical-interface>
    <physical-interface>
        <name>fe-4/0/0</name>
        <admin-status>up</admin-status>
        <oper-status>down</oper-status>
        <description>Customer B</description>
    </physical-interface>
</interface-information>
```

Output

```
Name: ge-0/0/0 Description: Management Interface
```

str:align()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

string **str:align**(string *[target]*, string *[padding]*)

string **str:align**(string *[target]*, string *[padding]*, string *[alignment]*)

Description

The **str:align**() function aligns a target string within a padding string and returns the result. The first argument is the target string, and the second argument is the padding string. The third argument is optional, and it specifies the type of alignment: “center”, “left”, or “right”. If the third argument is not present, or if it is not any of those three values, then the default alignment is to the left.

If the target string is longer than the padding string then the target string is returned, truncated to the length of the padding string. The truncation performed is always right truncation, regardless of what alignment type is specified.

Example

This op script demonstrates how the **str:align**() function can be used to format displayed text.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $title-padding =  "--------------------";
        var $column-padding = "                     ";

        /* print the title */
        <output> str:align( "Route", $title-padding, "left" ) _ " " _
                 str:align( "NH", $title-padding, "center" ) _ " " _
                 str:align( "Protocol", $title-padding, "right" );

        /* retrieve the routing table */
        var $route-rpc = {
            <get-route-information> {
                <table> "inet.0";
                <active-path>;
            }
        }
        var $routes = jcs:invoke( $route-rpc );

        /* Cycle through the routes */
        for-each( $routes/route-table/rt ) {
            <output> str:align( rt-destination, $column-padding ) _ " " _
                     str:align( rt-entry/nh/via, $column-padding, "center" ) _ " " _
                     str:align( rt-entry/protocol-name, $column-padding, "right" );
        }

    }
}
```

jcs:invoke() results

```
<route-information>
    <!-- keepalive -->
    <route-table>
        <table-name>inet.0</table-name>
        <destination-count>6</destination-count>
        <total-route-count>6</total-route-count>
        <active-route-count>6</active-route-count>
        <holddown-route-count>0</holddown-route-count>
        <hidden-route-count>0</hidden-route-count>
        <rt junos:style="brief">
            <rt-destination>0.0.0.0/0</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="5127">01:25:27</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.1</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.0/24</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Direct</protocol-name>
                <preference>0</preference>
                <age junos:seconds="5127">01:25:27</age>
                <nh>
                    <selected-next-hop/>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>10.0.0.10/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Local</protocol-name>
                <preference>0</preference>
                <age junos:seconds="5134">01:25:34</age>
                <nh-type>Local</nh-type>
                <nh>
                    <nh-local-interface>ge-0/0/0.0</nh-local-interface>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>192.168.0.0/16</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
```

```
                <last-active/>
                <protocol-name>Static</protocol-name>
                <preference>5</preference>
                <age junos:seconds="5127">01:25:27</age>
                <nh>
                    <selected-next-hop/>
                    <to>10.0.0.1</to>
                    <via>ge-0/0/0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>192.168.0.1/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>Direct</protocol-name>
                <preference>0</preference>
                <age junos:seconds="5207">01:26:47</age>
                <nh>
                    <selected-next-hop/>
                    <via>lo0.0</via>
                </nh>
            </rt-entry>
        </rt>
        <rt junos:style="brief">
            <rt-destination>224.0.0.5/32</rt-destination>
            <rt-entry>
                <active-tag>*</active-tag>
                <current-active/>
                <last-active/>
                <protocol-name>OSPF</protocol-name>
                <preference>10</preference>
                <age junos:seconds="5211">01:26:51</age>
                <metric>1</metric>
                <nh-type>MultiRecv</nh-type>
            </rt-entry>
        </rt>
    </route-table>
</route-information>
```

Output

```
Route--------------- ---------NH--------- ------------Protocol
0.0.0.0/0                 ge-0/0/0.0                   Static
10.0.0.0/24               ge-0/0/0.0                   Direct
10.0.0.10/32                                            Local
192.168.0.0/16            ge-0/0/0.0                   Static
192.168.0.1/32              lo0.0                      Direct
224.0.0.5/32                                             OSPF
```

str:concat()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str

Minimum Version: Junos 9.4

Syntax

string **str:concat**(node-set *[strings]*)

Description

The **str:concat**() function returns the string concatenation of all the string-values of the nodes of the node-set. If the node-set is empty then an empty string is returned.

Example

This op script demonstrates how the **str:concat**() function can combine multiple strings.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $number-set := {
            <node> "1";
            <node> "2";
            <node> "3";
            <node> "4";
            <node> "5";
        }

        var $letter-set := {
            <node> "a";
            <node> "b";
            <node> "c";
            <node> "d";
            <node> "e";
        }


        <output> str:concat( $number-set/node | $letter-set/node );
    }
}
```

Output

```
12345abcde
```

str:decode-uri()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

string **str:decode-uri**(string *[URI]*)

string **str:decode-uri**(string *[URI]*, string *[format]*)

Description

The **str:decode-uri**() function takes an encoded (escaped) URI string and returns it after converting all its escape sequences into UTF-8 characters. The second argument is optional, but if present can only be set to "UTF-8", as that is the only character format supported.

Example

This op script demonstrates how **str:decode-url**() can return the unescaped URI string.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";


match / {
    <op-script-results> {

        var $encoded-uri = "http://www.example.com/web%20pages/page%20%231/default.html";

        <output> str:decode-uri( $encoded-uri );
    }
}
```

Output

```
http://www.example.com/web pages/page #1/default.html
```

str:encode-uri()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

string **str:encode-uri**(string *[URI]*, boolean *[escape reserved?]*)

string **str:encode-uri**(string *[URI]*, boolean *[escape reserved?]*, string *[format]*)

Description

The **str:encode-uri**() function encodes a URI string by replacing special characters with escape sequences, following the encoding rules specified in RFC 2396 and amended in RFC 2732. The first string argument is the URI string that should be encoded. The second argument is a boolean that indicates whether or not reserved characters should be escaped. The third argument is an optional character encoding. When present it must be set to "UTF-8", which is also the default encoding if no third argument is present.

All characters will be escaped in the returned string except for the following:

- Upper and lower-case letters: A-Z, a-z
- Numbers: 0-9
- Marks (As defined by RFC 2396): - _ . ! ~ * ' ( )

If the escape reserved characters boolean argument is set to false then the following characters will also not be escaped:

- Reserved characters (As defined by RFC 2396 and RFC 2732): ; / ? : @ & = + $, [ ]

Note: As of the time of this writing, although the @ character is a reserved character, it is currently never escaped, even if the escape reserved characters boolean is set to true. Also, according to the EXSLT specification, the percent character % should not be escaped if it is followed by two hexadecimal digits, but **str:encode-uri**() currently always escapes it.

Because escaping reserved characters causes the forward slash and colon to be escaped, it generally makes sense to set the boolean argument to false when converting an entire URI, and to set it to true when converting only a part of a URI.

The **xf:escape-uri**() function is roughly equivalent to the **str:encode-uri**() function, except that **xf:escape-uri**() does not include the additional reserved characters added by RFC 2732.

Example

This op script demonstrates how **str:encode-uri**() can be used to encode a URI. The example shows the effect of both escaping and not escaping the reserved characters of a full URI string.

Code

```
version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";


match / {
    <op-script-results> {
        /* The original, unescaped URI */
        var $unencoded-uri = "http://www.example.com/web pages/page #1/default.html";
        <output> "Original: " _ $unencoded-uri;

        /* Encode it, but do not escape reserved characters */
        var $encoded-uri = str:encode-uri( $unencoded-uri, false() );
        <output> "Encoded: " _ $encoded-uri;

        /* Escape reserved characters too */
        var $all-escaped = str:encode-uri( $unencoded-uri, true()  );
        <output> "All escaped: " _ $all-escaped;

    }
}
```

Output

```
Original: http://www.example.com/web pages/page #1/default.html
Encoded: http://www.example.com/web%20pages/page%20%231/default.html
All escaped: http%3A%2F%2Fwww.example.com%2Fweb%20pages%2Fpage%20%231%2Fdefault.html
```

str:padding()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

string **str:padding**(number *[length]*)

string **str:padding**(number *[length]*, string)

Description

The **str:padding**() function creates a padding string of the length specified by the function's number argument. A padding string of spaces is created if no second argument is present; otherwise, the second argument is a string that is replicated as many times as necessary to create the desired size of padding string. If the string argument contains more than one character, then right truncation might be performed to ensure that the padding returned is the exact size required.

Example

This op script demonstrates how **str:padding**() can be used to create different types of padding strings.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Create some space padding */
        var $space-padding = str:padding( 10 );

        <output> "Space" _ $space-padding _ "padding" _ $space-padding _ "example";

        /* Create some string padding */
        var $padding = str:padding( 20, "=" );

        <output> $padding _ "Another Example" _ $padding;

    }
}
```

Output

```
Space          padding          example
====================Another Example====================
```

str:replace()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

string **str:replace**(string *[target]*, string *[search]*, string *[replacement]*)

string **str:replace**(string *[target]*, node-set *[search]*, string *[replacement]*)

string **str:replace**(string *[target]*, node-set *[search]*, node-set *[replacement]*)

Description

The **str:replace**() function replaces all occurrences of a string with another string and then returns the modified string. The first argument is the original string, the second argument is the search string or a node-set of search strings, and the third argument is the replacement string or a node-set of replacement strings.

If the search and replacement argument are both strings then the **str:replace**() function replaces each occurrence of the search string with the replacement string. If the replacement string is blank then the search string is deleted from the original string.

If the search argument is a node-set and the replacement argument is a string then the **str:replace**() function processes each node, in document order, of the search node-set by converting it to a string and then replacing any occurrences of its string value with the replacement string. If the replacement string is blank then that search string value is deleted from the original string.

If the search and replacement arguments are both node-sets then the **str:replace**() function processes each node, in document order, of the search node-set by converting it to a string and then replacing any occurrences of its string value with the string value of the node at the same index, in document order, of the replacement node-set. If the replacement string value is empty, or if there is not a node at that index of the replacement node-set, then the search string value is deleted from the original string.

Note: At the time of this writing, providing an empty search string will cause script processing to hang.

Example

This op script demonstrates the different ways that **str:replace**() can be used to modify strings.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $original-string = "From jnpr@host1 to jnpr@host5";

        /* Replace a string with another */
        <output> str:replace( $original-string, "jnpr", "admin" );

        /* Remove a string */
        <output> str:replace( $original-string, "jnpr@", "" );

        /* Replace multiple strings with one string */
        var $search-set := {
            <string> "host1";
            <string> "host5";
        }
        <output> str:replace( $original-string, $search-set/string, "new-host" );

        /* Replace multiple strings with multiple strings */
```

```
            var $replacement-set := {
                <string> "alpha";
                <string> "omega";
            }
            <output> str:replace( $original-string, $search-set/string, $replacement-set/string
);

            /* Unmatched search string gets deleted from string */
            var $search-set2 := {
                <string> "From";
                <string> "to";
            }
            var $replacement-set2 := {
                <string> "F:";
            }
            <output> str:replace( $original-string, $search-set2/string, $replacement-set2/string
);

        }
    }
```

Output

```
From admin@host1 to admin@host5
From host1 to host5
From jnpr@new-host to jnpr@new-host
From jnpr@alpha to jnpr@omega
F: jnpr@host1  jnpr@host5
```

str:split()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

node-set *[substrings]* **str:split**(string)

node-set *[substrings]* **str:split**(string *[target]*, string *[pattern]*)

Description

The **str:split**() function divides a string into substrings based on the location of a pattern string, and a node-set is returned with one node per substring. One or two string arguments are required. The first argument is the string that should be split. The second argument is optional; if it is present then it is the pattern string that the string should be split based on, but if it is not present then the default pattern string is a space (“ ”).

The complete pattern string is matched, but it is done in a case-insensitive manner. In contrast to the **jcs:split**() function, the pattern string of **str:split**() is not a regular expression.

If the pattern string is empty then the string is split into its component characters and each character is

returned as a separate node in the node-set. If the pattern string is not matched then the complete original string is returned within a single node.

Example

This op script demonstrates different uses of **str:split**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Use default space split */
        var $string1 = "one two three";
        var $tokens1 = str:split( $string1 );
        <output> "Split '" _ $string1 _ "' on ' '";
        for-each( $tokens1 ) {
            <output> .;
        }

        /* Split on ", " */
        var $string2 = "a, b, c, d";
        var $tokens2 = str:split( $string2, ", " );
        <output> "Split '" _ $string2 _ "' on ', '";
        for-each( $tokens2 ) {
            <output> .;
        }

        /* Split $hostname on every character */
        var $tokens3 = str:split( $hostname, "" );
        <output> "Split '" _ $hostname _ "' on ''";
        for-each( $tokens3 ) {
            <output> .;
        }

    }
}
```

Output

```
Split 'one two three' on ' '
one
two
three
Split 'a, b, c, d' on ', '
a
b
c
d
Split 'srx210' on ''
s
```

```
r
x
2
1
0
```

str:tokenize()

Source: EXSLT
Namespace: http://exslt.org/strings
Common Prefix: str
Minimum Version: Junos 9.4

Syntax

node-set *[substrings]* **str:tokenize**(string)

node-set *[substrings]* **str:tokenize**(string *[target]*, string *[delimiters]*)

Description

The **str:tokenize**() function divides a string into substrings based on the location of delimiters, and a node-set is returned with one node per substring. One or two string arguments are required. The first argument is the string that should be split into tokens. The second argument is optional; if it is present then it is a list of delimiter characters that the string should be split based on, but if it is not present then the default delimiter characters are the four white space characters:

- Space
- Tab - \t
- Newline - \n
- Carriage return - \r

The pattern matching behavior of **str:tokenize**() differs from that of **str:split**() because with **str:split**() the entire pattern string is matched, but with **str:tokenize**() each character in the delimiter string is treated as a potential delimiter. Also, unlike **str:split**(), the **str:tokenize**() function's delimiter matching is case-sensitive.

If the delimiter string is empty then the string is split into its component characters and each character is returned as a separate node in the node-set. If no delimiters are found in the original string, then it is returned unchanged, within a single node.

Example

This op script demonstrates different uses of **str:tokenize**().

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
```

```
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $version-string = jcs:sysctl( "kern.osrelease", "s" );

        /* Split out the version */
        var $version-tokens = str:tokenize( $version-string, "IRBS-" );

        /* Split the major and minor version */
        var $major-minor = str:tokenize( $version-tokens[1], "." );

        /* Character by character */
        var $characters = str:tokenize( $version-string, "" );

        <output> "Full version: " _ $version-string;
        <output> "Version: " _ $version-tokens[1];
        <output> "Major: " _ $major-minor[1];
        <output> "Minor: " _ $major-minor[2];

        <output> "Characters:";
        for-each( $characters ) {
            <output> .;
        }

        /* Whitespace example */
        var $example = "First Second\tThird\nFourth\rFifth";
        var $tokens = str:tokenize( $example );
        <output> "Whitespace Example: ";
        for-each( $tokens ) {
            <output> .;
        }
    }
}
```

Output

```
Full version: 10.3R1.9
Version: 10.3
Major: 10
Minor: 3
Characters:
1
0
.
3
R
1
.
9
Whitespace Example:
First
Second
Third
Fourth
Fifth
```

string()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **string**()

string **string**(object)

Description

The **string**() function returns the string value of the provided argument. If no argument is included then the **string**() function takes the context node as its argument. Data types are converted to string values in the following ways:

- boolean: A value of true is converted to "true" and a value of false is converted to "false".
- node-set: Converted to a string by returning the string-value of the first node of the node-set in document order. The string-value of a node is determined based on the node's type:
  - element node: string-value is the concatenation of all its descendant text nodes.
  - root node: string-value is the concatenation of all its descendant text nodes.
  - attribute node: string-value is the attribute's text value.
  - namespace node: string-value is the namespace URI.
  - processing instruction node: string-value is the text content that follows the instruction's target.
  - comment node: string-value is the comment text.
- number: Converted in the following manner:
  - NaN (Not-a-number) is converted to "NaN".
  - Positive infinity is converted to "Infinity".
  - Negative infinity is converted to "-Infinity".
  - All other numbers are converted into their string equivalent with a leading minus sign for negative numbers.
- result tree fragment: Converted to string by concatenating all the enclosed string values.

Example

This op script example shows the effect of the **string**() function called to convert a variety of data types.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
```

```
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $rtf = {
            <user> {
                <name> "lab";
                <class> "read-only";
            }
        }
        var $node-set := { copy-of $rtf; }
        var $true = true();
        var $false = false();
        var $integer = 12345;
        var $large-integer = 12345678901;
        var $decimal = -0.12345;
        var $infinity = 1 div 0;
        var $negative-infinity = -1 div 0;
        var $NaN = 0 div 0;

        <output> string( $rtf );
        <output> string( $node-set );
        <output> string( $node-set/user/name );
        <output> string( $true );
        <output> string( $false );
        <output> string( $integer );
        <output> string( $large-integer );
        <output> string( $decimal );
        <output> string( $infinity );
        <output> string( $negative-infinity );
        <output> string( $NaN );
    }
}
```

Output

```
labread-only
labread-only
lab
true
false
12345
 1.2345678901e+10
-0.12345
Infinity
-Infinity
NaN
```

string-length()

Source: XPath
Namespace: None
Common Prefix: None

Minimum Version: Junos 8.2

Syntax

number **string-length**()

number **string-length**(string)

Description

The **string-length**() function is used to determine the number of characters in a string. If no string argument is provided then it returns the number of characters in the string-value of the context node.

Example

This op script example demonstrates the use of **string-length**() to return the number of characters in some of the default parameters.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        <output> "$user: '" _ $user _ "' Length: " _ string-length( $user );
        <output> "$hostname: '" _ $hostname _ "' Length: " _ string-length( $hostname );
        <output> "$localtime: '" _ $localtime _ "' Length: " _ string-length( $localtime );
    }
}
```

Output

```
$user: 'jnpr' Length: 4
$hostname: 'srx210' Length: 6
$localtime: 'Fri Sep 10 09:28:56 2010' Length: 24
```

substring()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string *[substring]* **substring**(string, number *[start-index]*)

string *[substring]* **substring**(string, number *[start-index]*, number *[length]*)

Description

The **substring**() function returns a substring from within a string. Its arguments consist of the string from which the substring should be retrieved, as well as one or two number arguments. The first number argument is the starting index of the substring within the string. String indexes in SLAX begin at 1, so a starting index of 1 indicates that the substring should start with the first character, a starting index of 3 indicates that the substring should start with the third character, etc. The second number argument is not required, but if specified it indicates the number of characters, starting at the starting index, that should be retrieved. If the second number is not included then the function returns the substring that begins at the starting index and continues until the end of the string.

Example

This op script example shows how the **substring**() function can be used to retrieve various substrings from within a string based on the arguments provided.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $string = "ABCD.abcd";

        <output> substring( $string, 5 );
        <output> substring( $string, 1, 4 );
        <output> substring( $string, 6, 2 );
    }
}
```

Output

```
.abcd
ABCD
ab
```

substring-after()

Source: XPath

Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string *[substring]* **substring-after**(string *[target]*, string *[substring]*)

string *[substring]* **substring-after**(string *[target]*, string *[substring]*)

Description

The **substring-after**() function returns the substring from within the first string argument that immediately follows the second string argument. If the second string argument appears multiple times within the first string argument then the substring following the first occurrence of the second string is returned. If the second string argument does not appear within the first string argument then an empty string is returned.

Example

In this op script example, the **substring-after**() function is used to retrieve the build number from a version string.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $version = jcs:sysctl("kern.osrelease", "s");
    expr jcs:output( "Version String: ", $version );
    expr jcs:output( "Build Number: ", substring-after( $version, "R" ) );
}
```

Output

```
Version String: 10.0R4.7
Build Number: 4.7
```

substring-before()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string *[substring]* **substring-before**(string *[target]*, string *[substring]*)

string *[substring]* **substring-before**(string *[target]*, string *[substring]*)

Description

The **substring-before**() function returns the substring from within the first string argument that is before the second string argument. If the second string argument appears multiple times within the first string argument then the substring before the first occurrence of the second string is returned. If the second string argument does not appear within the first string argument then an empty string is returned.

Example

In this op script example, the **substring-before**() function is used to retrieve the major and minor version from a version string.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $version = jcs:sysctl("kern.osrelease", "s");
    expr jcs:output( "Version String: ", $version );
    expr jcs:output( "Major/Minor Version: ", substring-before( $version, "R" ) );
}
```

Output

```
Version String: 10.0R4.7
Major/Minor Version: 10.0
```

sum()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

number **sum**(node-set *[numbers]*)

Description

The **sum**() function returns the sum of all the nodes within a node-set. The string value of each node is first retrieved, then these string values are converted into numbers that are added together with the complete sum returned by the function.

Example

This op script example shows how the **sum**() function could be used to help determine the average file-system utilization on a Junos device.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $storage = jcs:invoke( "get-system-storage" );
        var $used-percent = $storage/filesystem/used-percent;
        var $average-percent = sum( $used-percent ) div count( $used-percent );
        <output> "Average used-percent = " _ $average-percent;
    }
}
```

jcs:invoke() results

```
<system-storage-information junos:style="brief">
    <filesystem>
        <filesystem-name>/dev/ad0s1a</filesystem-name>
        <total-blocks junos:format="217M">444092</total-blocks>
        <used-blocks junos:format="186M">379968</used-blocks>
        <available-blocks junos:format="14M">28600</available-blocks>
        <used-percent> 93</used-percent>
        <mounted-on>/</mounted-on>
    </filesystem>
    <filesystem>
        <filesystem-name>devfs</filesystem-name>
        <total-blocks junos:format="1.0K">2</total-blocks>
        <used-blocks junos:format="1.0K">2</used-blocks>
        <available-blocks junos:format="0B">0</available-blocks>
        <used-percent>100</used-percent>
        <mounted-on>/dev</mounted-on>
    </filesystem>
    <filesystem>
        <filesystem-name>devfs</filesystem-name>
        <total-blocks junos:format="1.0K">2</total-blocks>
        <used-blocks junos:format="1.0K">2</used-blocks>
        <available-blocks junos:format="0B">0</available-blocks>
        <used-percent>100</used-percent>
        <mounted-on>/dev/</mounted-on>
    </filesystem>
    <filesystem>
        <filesystem-name>/dev/md0</filesystem-name>
```

```
    <total-blocks junos:format="34M">69524</total-blocks>
    <used-blocks junos:format="34M">69524</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jbase</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md1</filesystem-name>
    <total-blocks junos:format="232M">475244</total-blocks>
    <used-blocks junos:format="232M">475244</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jkernel-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md2</filesystem-name>
    <total-blocks junos:format="60M">123060</total-blocks>
    <used-blocks junos:format="60M">123060</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jpfe-M320-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md3</filesystem-name>
    <total-blocks junos:format="5.5M">11268</total-blocks>
    <used-blocks junos:format="5.5M">11268</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jdocs-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md4</filesystem-name>
    <total-blocks junos:format="60M">123696</total-blocks>
    <used-blocks junos:format="60M">123696</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jroute-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md5</filesystem-name>
    <total-blocks junos:format="15M">30724</total-blocks>
    <used-blocks junos:format="15M">30724</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jcrypto-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md6</filesystem-name>
    <total-blocks junos:format="36M">73376</total-blocks>
    <used-blocks junos:format="36M">73376</used-blocks>
    <available-blocks junos:format="0B">0</available-blocks>
    <used-percent>100</used-percent>
    <mounted-on>/packages/mnt/jpfe-common-10.0R2.10</mounted-on>
</filesystem>
<filesystem>
    <filesystem-name>/dev/md7</filesystem-name>
    <total-blocks junos:format="2.0G">4122572</total-blocks>
    <used-blocks junos:format="10.0K">20</used-blocks>
    <available-blocks junos:format="1.8G">3792748</available-blocks>
    <used-percent>  0</used-percent>
    <mounted-on>/tmp</mounted-on>
</filesystem>
<filesystem>
```

```
            <filesystem-name>/dev/md8</filesystem-name>
            <total-blocks junos:format="2.0G">4122572</total-blocks>
            <used-blocks junos:format="1.6M">3280</used-blocks>
            <available-blocks junos:format="1.8G">3789488</available-blocks>
            <used-percent>  0</used-percent>
            <mounted-on>/mfs</mounted-on>
        </filesystem>
        <filesystem>
            <filesystem-name>/dev/ad0s1e</filesystem-name>
            <total-blocks junos:format="24M">48980</total-blocks>
            <used-blocks junos:format="26K">52</used-blocks>
            <available-blocks junos:format="22M">45012</available-blocks>
            <used-percent>  0</used-percent>
            <mounted-on>/config</mounted-on>
        </filesystem>
        <filesystem>
            <filesystem-name>procfs</filesystem-name>
            <total-blocks junos:format="4.0K">8</total-blocks>
            <used-blocks junos:format="4.0K">8</used-blocks>
            <available-blocks junos:format="0B">0</available-blocks>
            <used-percent>100</used-percent>
            <mounted-on>/proc</mounted-on>
        </filesystem>
        <filesystem>
            <filesystem-name>/dev/ad1s1f</filesystem-name>
            <total-blocks junos:format="25G">53051912</total-blocks>
            <used-blocks junos:format="6.6G">13935756</used-blocks>
            <available-blocks junos:format="17G">34872004</available-blocks>
            <used-percent> 29</used-percent>
            <mounted-on>/var</mounted-on>
        </filesystem>
    </system-storage-information>
```

Output

```
Average used-percent = 74.8
```

system-property()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

object **system-property** (string *[property]*)

Description

The **system-property**() function retrieves the system property value identified by the string argument, where "system" refers to the XSLT script processor, rather than the Junos operating system. There are three properties of the XSLT script processor that can be retrieved, all of which are returned as strings:

- xsl:vendor – The vendor of the XSLT script processor.

- xsl:vendor-url – A URL that corresponds to the vendor of the XSLT script processor.
- xsl:version – The XSLT version supported by the XSLT script processor.

Example

This op script shows the output of the three system properties that can be retrieved through the **system-property**() function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display system properties */
        <output> "xsl:vendor = " _ system-property( "xsl:vendor" );
        <output> "xsl:vendor-url = " _ system-property( "xsl:vendor-url" );
        <output> "xsl:version = " _ system-property( "xsl:version" );
    }
}
```

Output

```
xsl:vendor = libxslt
xsl:vendor-url = http://xmlsoft.org/XSLT/
xsl:version = 1.0
```

translate()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **translate**(string *[target]*, string *[matches]*, string *[replacements]*)

Description

The **translate**() function replaces characters within a string and returns the changed string. The second string contains all the characters that should be replaced. They are replaced by the characters in the same position within the third string. If there is no character in the third string at the same position, because the second string is longer than the third string, then that character will be removed rather than replaced.

Example

This op script example demonstrates how to use the **translate**() function to both replace and remove characters.

Code

```
    version 1.0;

    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match / {
        <op-script-results> {
            var $upper-user = translate( $user, "abcdefghijklmnopqrstuvwxyz",
"ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

            <output> "Username: " _ $user;
            <output> "Uppercase: " _ $upper-user;

            var $string = "Tab\tSpace Newline\nEnd";
            var $remove-whitespace = translate( $string, " \t\n\r", "" );

            <output> "String: " _ $string;
            <output> "Removed: " _ $remove-whitespace;
        }
    }
```

Output

```
    Username: jnpr
    Uppercase: JNPR
    String: Tab     Space Newline
    End
    Removed: TabSpaceNewlineEnd
```

true()

Source: XPath
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

boolean **true**()

Description

The **true**() function returns the boolean value of true.

Example

This op script example demonstrates how the **true**() function can be used to assign a boolean value of true to a template parameter.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        call display-boolean( $boolean = true() );
    }
}

template display-boolean( $boolean ) {
    if( $boolean ) {
        <output> "Boolean value is true";
    }
    if( not( $boolean ) ) {
        <output> "Boolean value is false";
    }
}
```

Output

```
Boolean value is true
```

unparsed-entity-uri()

Source: XSLT
Namespace: None
Common Prefix: None
Minimum Version: Junos 8.2

Syntax

string **unparsed-entity-uri**(string)

Description

The **unparsed-entity-uri**() function returns the string URI for an unparsed entity. This function is rarely used because the Junos XML API does not return unparsed entities in its results; however, it might be useful when processing an external XML document that was loaded by a Junos script.

Be aware that the **unparsed-entity-uri**() function searches within the XML document of the context node, so

the context node must be set to a node within the retrieved XML document, such as within a for-each loop, in order for the **unparsed-entity-uri**() function to find the unparsed entity.

Example

This is a simple op script example, showing how the **unparsed-entity-uri**() function can retrieve the URI for the specified unparsed entity.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    var $document = document( "/var/tmp/example.xml" );

    /* Change context to the loaded document */
    apply-templates $document/empty-example;
}

match empty-example {
    expr jcs:output( "Local Image URI is: ", unparsed-entity-uri( "local-image" ) );
}
```

example.xml

```
<?xml version="1.0"?>
<!DOCTYPE empty-example [
<!ENTITY local-image SYSTEM "file://var/tmp/image.gif" NDATA GIF>
<!ELEMENT empty-example EMPTY>
<!ATTLIST empty-example image ENTITY #REQUIRED>
]>
<empty-example image="local-image"/>
```

Output

```
Local Image URI is: file://var/tmp/image.gif
```

xf:escape-uri()

Source: XQuery
Namespace: http://www.w3.org/2002/08/xquery-functions
Common Prefix: xf
Minimum Version: Junos 8.2

Syntax

string **xf:escape-uri**(string *[URI]*, boolean *[escape reserved?]*)

Description

The **xf:escape-uri**() function encodes a URI string by replacing special characters with escape sequences, following the encoding rules specified in RFC 2396. The first argument is the URI string that should be encoded. The second argument is a boolean that indicates whether or not reserved characters should be escaped. Non-ascii characters are encoded per UTF-8 as part of the escape process.

All characters will be escaped in the returned string except for the following:

- Upper and lower-case letters: A-Z, a-z
- Numbers: 0-9
- Marks (As defined by RFC 2396): - _ . ! ~ * ‘ ( )

If the escape reserved characters boolean argument is set to false then the following characters will also not be escaped:

- Reserved characters (As defined by RFC 2396): ; / ? : @ & = + $,

In addition, the percent character % is only escaped if it is not followed by two hexadecimal digits.

Because escaping reserved characters causes the forward slash and colon to be escaped it generally makes sense to set the boolean argument to false when converting an entire URI, and to set it to true when converting only a part of a URI.

The **xf:escape-uri**() function is roughly equivalent to the **str:encode-uri**() function, except that **xf:escape-uri**() does not include the additional reserved characters added by RFC 2732.

Example

This op script demonstrates how **xf:escape-uri**() can be used to escape a URI. The example shows the effect of both escaping and not escaping the reserved characters of a full URI string.

Code

```
version 1.0;


ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns xf = "http://www.w3.org/2002/08/xquery-functions";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        /* The original, unescaped URI */
        var $unencoded-uri = "http://www.example.com/web pages/page #1/default.html";
        <output> "Original: " _ $unencoded-uri;

        /* Encode it, but do not escape reserved characters */
        var $encoded-uri = xf:escape-uri( $unencoded-uri, false() );
        <output> "Encoded: " _ $encoded-uri;

        /* Escape reserved characters too */
        var $all-escaped = xf:escape-uri( $unencoded-uri, true()  );
        <output> "All escaped: " _ $all-escaped;
```

```
    }
}
```

Output

```
Original: http://www.example.com/web pages/page #1/default.html
Encoded: http://www.example.com/web%20pages/page%20%231/default.html
All escaped: http%3A%2F%2Fwww.example.com%2Fweb%20pages%2Fpage%20%231%2Fdefault.html
```

xt:node-set()

Source: XT
Namespace: http://www.jclark.com/xt
Common Prefix: xt
Minimum Version: Junos 8.2

Syntax

node-set **xt:node-set**(node-set)

node-set **xt:node-set**(result-tree-fragment)

Description

The **xt:node-set**() function is used to convert a result tree fragment into a node-set. The behavior differs depending on the data type of the argument:

- Result tree fragment - A new XML document is created consisting of the nodes expressed by the result tree fragment. The returned node-set contains the root node of the new XML document.
- Node-set – The node-set argument is returned as the result.

One common stumbling block when using the xt:node-set() function is remembering that a converted result tree fragment results in a node-set that contains the root node of the new XML document rather than all of the XML children. For example, the following result tree fragment:

```
var $rtf = {
    <interface> "ge-0/0/0";
    <interface> "ge-0/0/1";
}
```

would be converted into a node-set with a single node: the root-node of the XML document, of which both of the <interface> nodes would be children; however, if a script writer wishes to have the node-set contain all the top-level child nodes rather than the root node then the following code can be used:

```
var $ns = xt:node-set( $rtf )/*;
```

The above example first converts the result tree fragment variable into a node-set; the returned root node is then subjected to a location-path to retrieve its children, and it is these child nodes that are assigned as a node-set to the $ns variable.

Note: The **libxslt:node-set**(), **saxon:node-set**(), and **xt:node-set**() functions are all identical. They also provide similar functionality as the **exsl:node-set**() function, except that the latter function is capable of converting strings, booleans, and numbers into node-sets as well.

Example

This op script demonstrates how the various node-set functions can be used both to convert a result tree fragment variable into a node-set as well as a string into a node-set.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns libxslt = "http://xmlsoft.org/XSLT/namespace";
ns saxon = "http://icl.com/saxon";
ns xt = "http://www.jclark.com/xt";
ns exsl = "http://exslt.org/common";

import "../import/junos.xsl";


match / {
    <op-script-results> {
        var $rtf = {
            <node> "one";
            <node> "two";
            <node> "three";
        }
        /* Convert RTF into node-set using all four methods */
        var $l-ns = libxslt:node-set( $rtf );
        var $s-ns = saxon:node-set( $rtf );
        var $x-ns = xt:node-set( $rtf );
        var $e-ns = exsl:node-set( $rtf );

        /* Display counts */
        <output> "libxslt #" _ count( $l-ns/node );
        <output> "saxon #" _ count( $s-ns/node );
        <output> "xt #" _ count( $x-ns/node );
        <output> "exsl #" _ count( $e-ns/node );

        /* Only exsl:node-set can convert string, booleans, and numbers */
        for-each( exsl:node-set( "Example text" ) ) {
            <output> "String converted by exsl:node-set() into text node: " _ .;
        }
    }
}
```

Output

```
libxslt #3
saxon #3
xt #3
exsl #3
String converted by exsl:node-set() into text node: Example text
```

Part 4: Elements

The underlying XSLT language differentiates between XML elements that are data and XML elements that are programming instructions, based on the element's namespace. The "xsl" namespace prefix is used to indicate a XSLT element, all of which are treated as part of the script's code rather than as part of its data. Many of these XSLT elements are replaced in the SLAX 1.0 language with programming statements, which are covered in the Statements section, but there are some XSLT elements that lack replacement SLAX statements, as well as some elements with capabilities that are unavailable through their equivalent statements, requiring that the XSLT elements be used within the SLAX script instead. This section covers the XSLT elements and additional extension elements that might be needed within a Junos SLAX script. Only elements that are processed as script code are included, meaning that result tree elements such as <output> or <xnm:error> and Junos API elements such as <get-configuration> or <file-put> are not covered.

The XSLT namespace prefix does not have to be defined within SLAX scripts because it is automatically added, with a prefix of "xsl", as part of the SLAX to XSLT conversion process when a SLAX script is executed; however, whenever a script uses an extension element, meaning that the element comes from a namespace other than XSLT, the element's namespace must be defined as an extension namespace. This is necessary to instruct the script processor that all elements from that namespace are script instructions, rather than XML data.

Each element is described as either an instruction or a top-level element, indicating where they can be used within a script; instruction elements can be used within a code block, such as a template or custom function code block, and top-level elements can be used at the top-level of the script, outside of any template or function.

All included attributes need to be assigned a value. There are three types of attributes, based on how their value is assigned and handled:

- Most attributes must have their value set as a hardcoded string. In this scenario, even numeric content must be enclosed within quotes (for example, name="element" size="10").
- Some attributes can have their value set through an attribute value template. This means that the attribute can have its value set as a hardcoded string, or it can be set based on the string value of a variable or XPath expression. In XSLT, these XPath expressions are surrounded by curly brackets: { }, but in SLAX the attribute assignment should be coded without quotes or curly brackets (e.g. href=$filename).
- The final type is an expression string, which is hardcoded within the script as a string, but is evaluated as a XPath expression. The assigned values are not attribute value templates, and so they must always be enclosed within quotes with literal string values enclosed in a double layer of quotes (for example, select="$filename" value="interfaces/interface" name="'name'"). Expression strings must be written using XSLT syntax only because the SLAX to XSLT conversion process does not convert them, so SLAX-specific operators cannot be used: "==","_", "||", "&&".

<exsl:document>

Source: EXSLT
Namespace: http://exslt.org/common
Common Prefix: exsl
Minimum Version: Junos 9.4

Syntax

<exsl:document append="*yes|no*" doctype-public="*string*" doctype-system="*string*"
encoding="*format*" href="*filename*" indent="*yes|no*" method="*html|text|xml*"
omit-xml-declaration="*yes|no*" standalone="*yes|no*" version="*string*" **>** {
File content...
}

Description

The **<exsl:document>** instruction element writes data to an output file on the local disk drive in XML, HTML, or text format. The filename is indicated through the required href attribute, which like all the other attributes of **<exsl:document>** is handled as an attribute value template, so its value can be specified as a hardcoded string, a variable, or a XPath expression.

It is recommended that the full path to the output file be provided because while the default directory for op scripts and commit scripts is /var/tmp, the default directory at times for event scripts is the root directory /, which cannot be written to by **<exsl:document>**. The only allowed URI scheme for the filename is "file", and it can be included as part of the href attribute value but is not required and there is no difference in behavior if it is not included (for example, href="file:///var/tmp/output").

The **<exsl:document>** element can only write to the local disk. Writing via networking protocols such as FTP or HTTP is not supported, and neither is writing to other routing-engines. Also, the full path specified in the href attribute must already exist because the script processor in Junos does not create new directories as part of the **<exsl:document>** operation.

The namespace for **<exsl:document>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns exsl extension = "http://exslt.org/common";
```

All contents of the **<exsl:document>** element are written to the output file:

Code example:

```
<exsl:document href="/var/tmp/example-file" indent="yes"> {
    <chassis> {
        <name> "M40e";
        <name> "EX8216";
    }
}
```

Output file:

```
<?xml version="1.0"?>
<chassis>
  <name>M40e</name>
  <name>EX8216</name>
</chassis>
```

The method attribute can be set to "xml", "html", or "text" and it determines the format for the output file.

The default method is “xml”; unless the root element of the output file is <html>, in which case the default method is “html”. When method is set to “text”, all of the text content is output, without any escaping or extra indenting.

Code example:

```
var $content = {
    <html> {
        <head> {
            <title> "Example HTML";
        }
        <body> {
            expr "First Line";
            <br>;
            expr "Second Line";
        }
    }
}
<exsl:document href="/var/tmp/xml-method" method="xml" indent="yes"> {
    copy-of $content;
}
<exsl:document href="/var/tmp/html-method" method="html"> {
    copy-of $content;
}
<exsl:document href="/var/tmp/text-method" method="text"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/xml-method
<?xml version="1.0"?>
<html>
  <head>
    <title>Example HTML</title>
  </head>
  <body>First Line<br/>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/html-method
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Example HTML</title>
</head>
<body>First Line<br>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/text-method
Example HTMLFirst LineSecond Line
```

By default, the **<exsl:document>** element overwrites the output file if it already exists; however, if the append attribute, which was added in Junos 11.1, is set to either “yes” or “true”, then the new data will be appended to the file, leaving the existing file contents in place. If append is set to “yes” or “true” when writing using the “xml” method, then the XML declaration is never added, even if the file is being created; however, at the time of this writing, if append is included and is not set to “yes” or “true” then the XML declaration is always added, even if omit-xml-declaration is set to “yes”, so it is best to only include the append attribute if it is being enabled, especially since it is off by default.

Code example:

```
<exsl:document href="/var/tmp/time-record" method="text"> {
    expr date:date-time() _ "\n";
```

```
}
<exsl:document href="/var/tmp/time-record" method="text" append="yes"> {
    expr date:date-time() _ "\n";
}
```

Output file:

```
2011-04-23T21:11:31Z
2011-04-23T21:11:31Z
```

A large caveat exists when using **<exsl:document>**; it always accesses the file system as user "nobody". This means that files cannot be overwritten or appended unless their permissions give write access to user "nobody" or to everyone, and files cannot be created unless the directory provides similar permissions. In addition, when the **<exsl:document>** element creates an output file, it is owned by user "nobody" and its file permissions are 644, meaning that while anyone can read it, only user "nobody" can edit or delete it. (As of Junos 10.0R3, super-users can delete these files from the CLI). As a workaround, create the file initially through the <file-put> RPC with permissions 666, giving everyone read and write access, and then have **<exsl:document>** overwrite the original file. As a result, the file will be owned by the executing script's user and will have the desired contents as created by **<exsl:document>**.

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
        }
    }
}
<exsl:document href="/var/tmp/destination-config" indent="yes"> {
    copy-of jcs:invoke( $rpc );
}
```

Output:

```
jnpr@srx210> file list /var/tmp/destination-config detail
-rw-r--r--  1 nobody wheel       405 Apr 23 22:01 /var/tmp/destination-config
total 1

jnpr@srx210> file show /var/tmp/destination-config
<?xml version="1.0"?>
<configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
    <event-options>
        <destinations>
            <name>local</name>
            <archive-sites>
                <name>/var/tmp</name>
            </archive-sites>
        </destinations>
    </event-options>
</configuration>
```

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
        }
    }
```

```
    }
    var $filename = "/var/tmp/destination-config";
    var $put-rpc = {
        <file-put> {
            <filename> $filename;
            <permission> "666";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "To be overwritten";
        }
    }
    var $results = jcs:invoke( $put-rpc );
    <exsl:document href=$filename indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-rw-rw-  1 jnpr  staff        405 Apr 23 22:07 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in the output file:

Code example:

```
    <exsl:document href="/var/tmp/output" doctype-system="local.dtd"> {
        <parent> {
            <child>;
            <child>;
            <child>;
        }
    }
```

Output file:

```
    <?xml version="1.0"?>
    <!DOCTYPE parent SYSTEM "local.dtd">
    <parent><child/><child/><child/></parent>
```

The encoding attribute determines what character encoding format should be used when writing the output file. The following (case-insensitive) values can be used: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii. The default value is utf-8.

Code example:

```
    <exsl:document href="/var/tmp/output-html" encoding="html"> {
        <copyright> "©";
    }
```

Output file:

```
<?xml version="1.0" encoding="html"?>
<copyright>&copy;</copyright>
```

The indent attribute can be set to either “yes” or “no”, and it determines whether indentation should be performed or not. It is off by default for the “xml” method and on by default for the “html” method. It has no effect on text files.

Code example:

```
var $content = {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
        }
    }
}
<exsl:document href="/var/tmp/output1" method="xml"> {
    copy-of $content;
}
<exsl:document href="/var/tmp/output2" method="xml" indent="yes"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/output1
<?xml version="1.0"?>
<interfaces><interface><name>ge-0/0/0</name></interface></interfaces>

jnpr@srx210> file show /var/tmp/output2
<?xml version="1.0"?>
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
  </interface>
</interfaces>
```

A XML declaration is included by default when using the “xml” method (unless appending is enabled). To disable the XML declaration, set the omit-xml-declaration attribute to “yes”. This attribute has no effect when using the “html” or “text” methods.

Code example:

```
<exsl:document href="/var/tmp/output" indent="yes"
               omit-xml-declaration="yes"> {
    <details> {
        <script> $script;
        <user> $user;
    }
}
```

Output file:

```
<details>
  <script>test.slax</script>
  <user>jnpr</user>
</details>
```

By default, no standalone document declaration is included in the XML declaration, but if the standalone attribute is set to either “yes” or “no”, then standalone is included in the declaration with the specified value.

Code example:

```
<exsl:document href="/var/tmp/context" indent="yes" standalone="yes" > {
    copy-of $junos-context/user-context;
}
```

Output file:

```
<?xml version="1.0" standalone="yes"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

The version attribute is used to alter the XML version value that is written in the XML declaration of the output file when the "xml" method is used. It should be set to "1.0", but changing it to other values does not alter the format of the document in any way.

The **<exsl:document>** element has an additional attribute: cdata-section-elements, which is intended to indicate what nodes in the XML output file should have their text contents enclosed within CDATA sections, but it is non-functional at the time of this writing. Also, the media-type attribute is not implemented and has no effect.

If an error occurs due to an inability to write to the output file, an incorrect attribute setting, or for some other reason, then the script fails and an error message is displayed to the script user for op scripts and commit scripts, or to the event script output file. The script itself is unable to react to the error.

The Junos API contains a RPC element for writing to the disk: <file-put>, which has some capabilities that **<exsl:document>** does not have, including the ability to write in base64 format, to select file permissions, and to write to the non-local routing-engines. In addition, unlike **<exsl:document>**, the <file-put> RPC accesses the file system using the credentials of the user that is executing the script. However, <file-put> is incapable of writing in XML or HTML format or of appending to an existing file, so when a script needs to perform one of those three operations, then **<exsl:document>** would be an appropriate element to use.

Junos contains support for five instruction elements that have equivalent functionality: **<exsl:document>**, **<redirect:write>**, **<saxon:output>**, **<xsl:document>**, and **<xt:document>. All attributes and functionality are common among the five elements, except for some variances in support for the different file selector attributes: file, href, and select. Given that all five elements perform the same operation, it is recommended that the <xsl:document> element be used in place of the others because, unlike the other four, it does not need to have an extension namespace defined.**

Example

This op script demonstrates how to use the five file writing instruction elements.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl extension = "http://exslt.org/common";
ns saxon extension = "http://icl.com/saxon";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
ns xt extension = "http://www.jclark.com/xt";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* <redirect:write> <xsl:document> <saxon:output> and appending */
        var $append-file = "/var/tmp/append-example";
        <redirect:write href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <xsl:document href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <saxon:output href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }

        /* <exsl:document> and indenting */
        <exsl:document href="/var/tmp/indent-example" indent="yes"> {
            <chassis> {
                <platform> "m40e";
                <platform> "mx960";
                <platform> "ex8216";
            }
        }

        /* <xt:document> and text output */
        <xt:document href="/var/tmp/text-example" method="text"> {
            expr "Class-name: " _ $junos-context/user-context/class-name _ "\n";
            expr "User-name: " _ $junos-context/user-context/user _ "\n";
        }
    }
}
```

Output

```
jnpr@srx210> file list detail /var/tmp/*-example
-rw-r--r--  1 nobody wheel       318 Apr 25 15:31 /var/tmp/append-example
-rw-r--r--  1 nobody wheel       130 Apr 25 15:31 /var/tmp/indent-example
-rw-r--r--  1 nobody wheel        41 Apr 25 15:31 /var/tmp/text-example
total 3

jnpr@srx210> file show /var/tmp/append-example
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>

jnpr@srx210> file show /var/tmp/indent-example
<?xml version="1.0"?>
<chassis>
  <platform>m40e</platform>
  <platform>mx960</platform>
  <platform>ex8216</platform>
</chassis>

jnpr@srx210> file show /var/tmp/text-example
Class-name: j-super-user
User-name: jnpr
```

<func:function>

Source: EXSLT
Namespace: http://exslt.org/functions
Common Prefix: func
Minimum Version: Junos 9.4

Syntax

<func:function name=*"function-name"* **>** {

Parameters...
Function code...
}

Description

The **<func:function>** top-level element is used to create a custom function, which can be invoked in the same way as the standard SLAX functions. Its name attribute is required and indicates the name of the function. It must be set as a hardcoded string and should be a valid qualified name (see Appendix B). The name must contain a namespace prefix, which should be associated with a custom namespace. The content of the **<func:function>** element is the function's code, and it is processed every time the function is invoked.

Code example:

```
match / {
    <op-script-results> {
        expr example:print-message();
    }
}

<func:function name="example:print-message"> {
    expr jcs:output("Hello World!");
}
```

Output:

```
Hello World!
```

Function arguments are added by including parameters within the function's code. The order of these parameters is significant, as function arguments are assigned by order, rather than by name, like template parameters. Default values can be provided for function arguments by assigning them to the parameters. Two functions with the same name cannot occur within the same script, but duplicate function names can exist if one is imported from an import file. The function with the higher import precedence is always used; function overloading (selecting the function based on the arguments) is not supported.

Code example:

```
match / {
    <op-script-results> {
        expr example:output-reverse("Hello World!");
    }
}

<func:function name="example:output-reverse"> {
    param $string;

    var $chars = str:tokenize( $string, "" );
```

```
    var $new-string = {
        for-each( $chars ) {
            <xsl:sort data-type="number" order="descending"
                      select="position()">;
            expr .;
        }
    }
    expr jcs:output( $new-string );
}
```

Output:

```
!dlroW olleH
```

The namespace for **<func:function>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an extension element:

```
ns func extension = "http://exslt.org/functions";
```

In addition, the custom namespace used for the function name must be defined as well, but does not need to be an extension namespace. (*Note:* script writers should use a URL that they own or that their organization owns.):

```
ns example = "http://xml.juniper.net/example";
```

It is illegal to write to the result tree within a function; instead, any function results should be returned through the **<func:result>** instruction element:

Code example:

```
match / {
    <op-script-results> {
        <output> example:is-letters( "5A" );
        <output> example:is-letters( "100.1" );
        <output> example:is-letters( "Test!" );
        <output> example:is-letters( "ABCDEFG" );
    }
}

<func:function name="example:is-letters"> {
    param $string;
    var $result = jcs:regex( "[^[:alpha:]]+", $string );

    <func:result select="jcs:empty( $result )">;
}
```

Output:

```
false
false
false
true
```

Example

This op script demonstrates how to use **<func:function>** to define a custom function.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
ns func extension = "http://exslt.org/functions";
ns example = "http://xml.juniper.net/example";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $input = jcs:get-input("Enter string: " );
        <output> "In lower case: " _ example:to-lower( $input );
    }
}

<func:function name="example:to-lower"> {
    param $string;

    <func:result select=
        "translate($string,'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')">;
}
```

Output

```
jnpr@srx210> op func_function
Enter string: Juniper Networks
In lower case: juniper networks
```

<func:result>

Source: EXSLT
Namespace: http://exslt.org/functions
Common Prefix: func
Minimum Version: Junos 9.4

Syntax

<func:result select= "*result expression* ">;

<func:result> {
  RTF result value...
}

Description

The **<func:result>** instruction element is used within a custom function to select what the function's result should be. The result indicated by **<func:result>** can be any data type: boolean, external, node-set, number, result tree fragment, or string.

A result can be selected in one of three ways:

- If the select attribute is present then its value is evaluated as a XPath expression and the result of that expression, which could be any data type, becomes the function's result. The select attribute is a hard-coded expression string, and only standard XSLT syntax can be used in the string; SLAX-specific operators are not permitted. (If the result is a string literal then it must be enclosed within a double-layer of quotes to ensure it is not evaluated as a location path. For example: select=" 'admin' ").

Code example:

```
match / {
    <op-script-results> {
        var $result = example:is-odd( 101 );
        <output> exsl:object-type( $result );
        <output> $result;
    }
}

<func:function name="example:is-odd"> {
    param $number;

    /* SLAX == operator cannot be used */
    <func:result select="($number mod 2) = 1)">;
}
```

Output:

```
boolean
true
```

- If the select attribute is not present and instead child contents are assigned to the **<func:result>** element then the function's result is a result tree fragment consisting of the contents of **<func:result>**.

Code example:

```
match / {
    <op-script-results> {
        var $result = example:extract-vowels( "abcdefghij" );
        <output> exsl:object-type( $result );
        var $result-ns = exsl:node-set( $result );
        for-each( $result-ns/vowel ) {
            <output> .;
        }
    }
}

<func:function name="example:extract-vowels"> {
    param $string;

    /* Break into characters */
    var $characters = str:tokenize( $string, "" );

    <func:result> {
        for-each( $characters ) {
            /* Check if it is a vowel */
            var $check = translate( ., "AEIOUYaeiouy", "" );
            /* if it is blank then the current char is a vowel */
            if( string-length( $check ) == 0 ) {
                <vowel> .;
            }
        }
    }
}
```

Output:

```
RTF
a
e
i
```

- If the **<func:result>** element has no select attribute or child contents then an empty string is the function's result.

An error is generated, causing the script to fail, if a **<func:result>** element has both a select attribute as well as child contents. The **<func:result>** element only selects the function's result. Unlike the "return" statement in other programming languages, it does not cause the function processing to cease, but no other SLAX statements or instruction elements are allowed to be processed within the function after the **<func:result>** element (other than **<xsl:fallback>**), so the function's code must be structured to ensure that **<func:result>** is the final instruction processed:

Code example:

```
match / {
    <op-script-results> {
        <output> example:slot-count( "MX960" );
        <output> example:slot-count( "MX480" );
        <output> example:slot-count( "MX240" );
        <output> example:slot-count( "?" );
    }
}

<func:function name="example:slot-count"> {
    param $chassis;

    /* In all cases - <func:result> is processed last */
    if( $chassis == "MX960" ) {
        <func:result select="12">;
    }
    else if( $chassis == "MX480" ) {
        <func:result select="6">;
    }
    else if( $chassis == "MX240" ) {
        <func:result select="3">;
    }
    else {
        /* Double-layer of quotes because it is an actual string
           instead of a location path */
        <func:result select="'Unknown'">;
    }
}
```

Output:

```
12
6
3
Unknown
```

The namespace for **<func:result>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns func extension = "http://exslt.org/functions";
```

Example

This op script demonstrates how to use **<func:result>** within a custom function to indicate what the function's result should be.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
ns func extension = "http://exslt.org/functions";
ns example = "http://xml.juniper.net/example";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        var $input = jcs:get-input("Enter string: " );
        <output> "In lower case: " _ example:to-lower( $input );
    }
}

<func:function name="example:to-lower"> {
    param $string;

    <func:result select=
        "translate($string,'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')">;
}
```

Output

```
jnpr@srx210> op func_function
Enter string: Juniper Networks
In lower case: juniper networks
```

<libxslt:debug>

Source: Libxslt
Namespace: http://xmlsoft.org/XSLT/namespace
Common Prefix: libxslt
Minimum Version: Junos 8.2

Syntax

<libxslt:debug>;

Description

The **<libxslt:debug>** instruction element provides basic debugging information from the running script. In particular, it displays the backtrace of the templates in operation, as well as the names of the variables and parameters that are defined within those templates.

The information provided by **<libxslt:debug>** is displayed to the CLI user of op scripts and commit scripts and is handled as an error message, meaning that its use will always cause commit scripts to fail the commit. The information is also written to the trace file of the particular script type.

The template names are displayed starting from the current template and working backward. They are identified by either their template name, or if they are match templates by their match pattern, but both of these values are displayed following the word “name”

Displayed to the committing user:

```
error: Templates:
error: #0
error: name called-template
```

```
error: #1
error: name configuration
error: #2
error: name /
```

Seen in the commit script trace file:

```
Apr 14 15:02:30 Templates:
Apr 14 15:02:30 #0
Apr 14 15:02:30 name called-template
Apr 14 15:02:30
Apr 14 15:02:30 #1
Apr 14 15:02:30 name configuration
Apr 14 15:02:30
Apr 14 15:02:30 #2
Apr 14 15:02:30 name /
Apr 14 15:02:30
```

The names of all the template variables and parameters are displayed as well, starting with the current template and working backward through all the templates being processed, but unfortunately neither the variable's data type nor its value are displayed in Junos, making this output section less valuable than the template backtrace.

Whether the variable is a variable, or a parameter, it is displayed> However, parameters are not displayed correctly when they have been set by the calling template (as opposed to allowing their default values to be applied).

Code example:

```
match configuration {
    var $interface-hierarchy = interfaces;
    call called-template();
}
template called-template( $interface="ge-0/0/0", $unit="10" ) {
    var $new-name = "ge-0/0/1";
    <libxslt:debug>;
}
```

Variables portion of output:

```
error: Variables:
error: #0
error: var
error: new-name
error: #1
error: param
error: unit
error: #2
error: param
error: interface
error: #3
error: var
error: interface-hierarchy
```

When a match template is called with a mode specified then the mode will be displayed following the match pattern of the template:

Code example:

```
match configuration {
   apply-templates interfaces {
      mode "default";
   }
}
match interfaces {
```

```
    mode “default”;
    <libxslt:debug>;
}
```

Templates portion of output:

```
error: Templates:
error: #0
error: name interfaces
error: name default
error: #1
error: name configuration
error: #2
error: name /
```

The namespace for **<libxslt:debug>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns libxslt extension = "http://xmlsoft.org/XSLT/namespace";
```

The debugging information provided by **<libxslt:debug>** is inferior to that available through the script debugger, but the element might be useful if a template backtrace is desired within a commit or event script, or if it is desired prior to Junos 10.4, when the script debugger was first introduced.

Example

This op script demonstrates how **<libxslt:debug>** can be used to retrieve the template backtrace information.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns libxslt extension = "http://xmlsoft.org/XSLT/namespace";

import "../import/junos.xsl";

match / {
    call template-1();
}

template template-1() {
    call template-2();
}

template template-2() {
    call template-3();
}

template template-3() {
    call template-4();
}

template template-4() {
    call template-5();
}

template template-5() {
    <libxslt:debug>;
}
```

Output

```
error: Templates:
error: #0
error: name template-5
error: #1
error: name template-4
error: #2
error: name template-3
error: #3
error: name template-2
error: #4
error: name template-1
error: #5
error: name /
error: Variables:
```

<redirect:write>

Source: XALAN
Namespace: org.apache.xalan.xslt.extensions.Redirect
Common Prefix: redirect
Minimum Version: Junos 8.2

Syntax

<redirect:write append=*"yes|no"* doctype-public=*"string"* doctype-system=*"string"*

encoding=*"format"* file=*"filename"* href=*"filename"* indent=*"yes|no"*

method=*"html|text|xml"* omit-xml-declaration=*"yes|no"*

select=*"filename"* standalone=*"yes|no"* version=*"string"* **>** {

File content...

}

Description

The **<redirect:write>** instruction element writes data to an output file on the local disk drive in XML, HTML, or text format. The filename can be indicated through the select attribute, which is a hardcoded expression string that is executed as a XPath expression to determine the actual filename; otherwise, it can be set by the file or href attributes, both of which are handled as attribute value templates (like all other attributes of **<redirect:write>**), so their values can be specified as a hardcoded string, a variable, or a XPath expression. (for example select="'/var/tmp/output'", file="/var/tmp/output", and href="/var/tmp/output" are all equivalent, as are select="$filename", file=$filename, and href=$filename. Note the double-layer of quotes around the first select example as well as the quotes in the second select example).

It is recommended that the full path to the output file be provided, because while the default directory for op scripts and commit scripts is /var/tmp, the default directory at times for event scripts is the root directory /, which cannot be written to by **<redirect:write>**. The only allowed URI scheme for the filename is "file", and it can be included as part of the href attribute value, but is not required, and there is no difference in behavior if

it is not included (for example, href="file:///var/tmp/output").

The **<redirect:write>** element can only write to the local disk. Writing via networking protocols such as FTP or HTTP is not supported, and neither is writing to other routing-engines. Also, the full path specified in the href attribute (or select or file) must already exist because the script processor in Junos does not create new directories as part of the **<redirect:write>** operation.

The namespace for **<redirect:write>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
All contents of the <redirect:write> element are written to the output file:
```

Code example:

```
<redirect:write href="/var/tmp/example-file" indent="yes"> {
    <chassis> {
        <name> "M40e";
        <name> "EX8216";
    }
}
```

Output file:

```
<?xml version="1.0"?>
<chassis>
  <name>M40e</name>
  <name>EX8216</name>
</chassis>
```

The method attribute can be set to “xml”, “html”, or “text” and it determines the format for the output file. The default method is “xml”; unless the root element of the output file is <html>, in which case the default method is “html”. When method is set to “text”, all of the text content is output, without any escaping or extra indenting.

Code example:

```
var $content = {
    <html> {
        <head> {
            <title> "Example HTML";
        }
        <body> {
            expr "First Line";
            <br>;
            expr "Second Line";
        }
    }
}
<redirect:write href="/var/tmp/xml-method" method="xml" indent="yes"> {
    copy-of $content;
}
<redirect:write href="/var/tmp/html-method" method="html"> {
    copy-of $content;
}
<redirect:write href="/var/tmp/text-method" method="text"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/xml-method
<?xml version="1.0"?>
<html>
  <head>
```

```
    <title>Example HTML</title>
  </head>
  <body>First Line<br/>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/html-method
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Example HTML</title>
</head>
<body>First Line<br>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/text-method
Example HTMLFirst LineSecond Line
```

By default, the **<redirect:write>** element overwrites the output file if it already exists; however, if the append attribute, which was added in Junos 11.1, is set to either "yes" or "true", then the new data will be appended to the file, leaving the existing file contents in place. If append is set to "yes" or "true" when writing using the "xml" method, then the XML declaration is never added, even if the file is being created; however, at the time of this writing, if append is included and is not set to "yes" or "true" then the XML declaration is always added, even if omit-xml-declaration is set to "yes", so it is best to only include the append attribute if it is being enabled, especially since it is off by default.

Code example:

```
<redirect:write href="/var/tmp/time-record" method="text"> {
    expr date:date-time() _ "\n";
}
<redirect:write href="/var/tmp/time-record" method="text" append="yes"> {
    expr date:date-time() _ "\n";
}
```

Output file:

```
2011-04-23T21:11:31Z
2011-04-23T21:11:31Z
```

A large caveat exists when using **<redirect:write>**: it always accesses the file system as user "nobody". This means that files cannot be overwritten or appended unless their permissions give write access to user "nobody" or to everyone, and files cannot be created unless the directory provides similar permissions. In addition, when the **<redirect:write>** element creates an output file, it is owned by user "nobody" and its file permissions are 644, meaning that while anyone can read it, only user "nobody" can edit or delete it. (As of Junos 10.0R3, super-users can delete these files from the CLI). As a workaround, create the file initially through the <file-put> RPC with permissions 666, giving everyone read and write access, and then have **<redirect:write>** overwrite the original file. As a result, the file will be owned by the executing script's user and will have the desired contents as created by **<redirect:write>**.

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
        }
    }
}
<redirect:write href="/var/tmp/destination-config" indent="yes"> {
    copy-of jcs:invoke( $rpc );
```

```
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-r--r--  1 nobody wheel       405 Apr 23 22:01 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

Example code:

```
    var $rpc = {
        <get-configuration> {
            <configuration> {
                <event-options> {
                    <destinations>;
                }
            }
        }
    }
    var $filename = "/var/tmp/destination-config";
    var $put-rpc = {
        <file-put> {
            <filename> $filename;
            <permission> "666";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "To be overwritten";
        }
    }
    var $results = jcs:invoke( $put-rpc );
    <redirect:write href=$filename indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-rw-rw-  1 jnpr  staff        405 Apr 23 22:07 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
```

```
    </event-options>
</configuration>
```

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in the output file:

Code example:

```
<redirect:write href="/var/tmp/output" doctype-system="local.dtd"> {
    <parent> {
        <child>;
        <child>;
        <child>;
    }
}
```

Output file:

```
<?xml version="1.0"?>
<!DOCTYPE parent SYSTEM "local.dtd">
<parent><child/><child/><child/></parent>
```

The encoding attribute determines what character encoding format should be used when writing the output file. The following (case-insensitive) values can be used: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii. The default value is utf-8.

Code example:

```
<redirect:write href="/var/tmp/output-html" encoding="html"> {
    <copyright> "©";
}
```

Output file:

```
<?xml version="1.0" encoding="html"?>
<copyright>&copy;</copyright>
```

The indent attribute can be set to either “yes” or “no”, and it determines whether indentation should be performed or not. It is off by default for the “xml” method and on by default for the “html” method. It has no effect on text files.

Code example:

```
var $content = {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
        }
    }
}
<redirect:write href="/var/tmp/output1" method="xml"> {
    copy-of $content;
}
<redirect:write href="/var/tmp/output2" method="xml" indent="yes"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/output1
<?xml version="1.0"?>
<interfaces><interface><name>ge-0/0/0</name></interface></interfaces>

jnpr@srx210> file show /var/tmp/output2
```

```
<?xml version="1.0"?>
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
  </interface>
</interfaces>
```

A XML declaration is included by default when using the “xml” method (unless appending is enabled). To disable the XML declaration, set the omit-xml-declaration attribute to “yes”. This attribute has no effect when using the “html” or “text” methods.

Code example:

```
<redirect:write href="/var/tmp/output" indent="yes"
                omit-xml-declaration="yes"> {
    <details> {
        <script> $script;
        <user> $user;
    }
}
```

Output file:

```
<details>
  <script>test.slax</script>
  <user>jnpr</user>
</details>
```

By default, no standalone document declaration is included in the XML declaration, but if the standalone attribute is set to either “yes” or “no”, then standalone is included in the declaration with the specified value.

Code example:

```
<redirect:write href="/var/tmp/context" indent="yes" standalone="yes" > {
    copy-of $junos-context/user-context;
}
```

Output file:

```
<?xml version="1.0" standalone="yes"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

The version attribute is used to alter the XML version value that is written in the XML declaration of the output file when the “xml” method is used. It should be set to “1.0”, but changing it to other values does not alter the format of the document in any way.

The **<redirect:write>** element has an additional attribute: cdata-section-elements, which is intended to indicate what nodes in the XML output file should have their text contents enclosed within CDATA sections, but it is non-functional at the time of this writing. Also, the media-type attribute is not implemented and has no effect.

If an error occurs due to an inability to write to the output file, an incorrect attribute setting, or for some other reason, then the script fails and an error message is displayed to the script user for op scripts and commit scripts or to the event script output file. The script itself is unable to react to the error.

The Junos API contains a RPC element for writing to the disk: <file-put>, which has some capabilities that **<redirect:write>** does not have, including the ability to write in base64 format, to select file permissions, and to write to the non-local routing-engines. In addition, unlike **<redirect:write>**, the <file-put> RPC accesses the file system using the credentials of the user that is executing the script. However, <file-put> is incapable of writing

in XML or HTML format or of appending to an existing file, so when a script needs to perform one of those three operations then **<redirect:write>** would be an appropriate element to use.

Junos contains support for five instruction elements that have equivalent functionality: **<exsl:document>**, **<redirect:write>**, **<saxon:output>**, **<xsl:document>**, and **<xt:document>. All attributes and functionality are common among the five elements, except for some variances in support for the different file selector attributes: file, href, and select. Given that all five elements perform the same operation, it is recommended that the <xsl:document> element be used in place of the others because, unlike the other four, it does not need to have an extension namespace defined.**

Example

This op script demonstrates how to use the five file writing instruction elements.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl extension = "http://exslt.org/common";
ns saxon extension = "http://icl.com/saxon";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
ns xt extension = "http://www.jclark.com/xt";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* <redirect:write> <xsl:document> <saxon:output> and appending */
        var $append-file = "/var/tmp/append-example";
        <redirect:write href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <xsl:document href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <saxon:output href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }

        /* <exsl:document> and indenting */
        <exsl:document href="/var/tmp/indent-example" indent="yes"> {
            <chassis> {
                <platform> "m40e";
                <platform> "mx960";
                <platform> "ex8216";
            }
        }

        /* <xt:document> and text output */
        <xt:document href="/var/tmp/text-example" method="text"> {
            expr "Class-name: " _ $junos-context/user-context/class-name _ "\n";
            expr "User-name: " _ $junos-context/user-context/user _ "\n";
        }
    }
}
```

Output

```
    jnpr@srx210> file list detail /var/tmp/*-example
    -rw-r--r--  1 nobody wheel       318 Apr 25 15:31 /var/tmp/append-example
    -rw-r--r--  1 nobody wheel       130 Apr 25 15:31 /var/tmp/indent-example
    -rw-r--r--  1 nobody wheel        41 Apr 25 15:31 /var/tmp/text-example
    total 3

    jnpr@srx210> file show /var/tmp/append-example
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>

    jnpr@srx210> file show /var/tmp/indent-example
    <?xml version="1.0"?>
    <chassis>
      <platform>m40e</platform>
      <platform>mx960</platform>
      <platform>ex8216</platform>
    </chassis>

    jnpr@srx210> file show /var/tmp/text-example
    Class-name: j-super-user
    User-name: jnpr
```

<saxon:output>

Source: SAXON
Namespace: http://icl.com/saxon
Common Prefix: saxon
Minimum Version: Junos 8.2

Syntax

<saxon:output append=*"yes|no"* doctype-public=*"string"* doctype-system=*"string"*

encoding=*"format"* file=*"filename"* href=*"filename"* indent=*"yes|no"*

method=*"html|text|xml"* omit-xml-declaration=*"yes|no"*

standalone=*"yes|no"* version=*"string"* **>** {

File content...

}

Description

The **<saxon:output>** instruction element writes data to an output file on the local disk drive in XML, HTML, or text format. The filename must be indicated through the file or href attributes, both of which are handled as attribute value templates (like all other attributes of **<saxon:output>**), so their values can be specified as a hardcoded string, a variable, or a XPath expression.

It is recommended that the full path to the output file be provided, because while the default directory for op scripts and commit scripts is /var/tmp, the default directory at times for event scripts is the root directory /, which cannot be written to by **<saxon:output>**. The only allowed URI scheme for the filename is "file", and it can be included as part of the href attribute value, but is not required, and there is no difference in behavior if it is not included (e.g. href="file:///var/tmp/output").

The **<saxon:output>** element can only write to the local disk. Writing via networking protocols such as FTP or HTTP is not supported, and neither is writing to other routing-engines. Also, the full path specified in the href attribute (or select or file) must already exist because the script processor in Junos does not create new directories as part of the **<saxon:output>** operation.

The namespace for **<saxon:output>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns saxon extension = "http://icl.com/saxon";
```

All contents of the <saxon:output> element are written to the output file:

Code example:

```
<saxon:output href="/var/tmp/example-file" indent="yes"> {
    <chassis> {
        <name> "M40e";
        <name> "EX8216";
    }
}
```

Output file:

```
<?xml version="1.0"?>
<chassis>
  <name>M40e</name>
  <name>EX8216</name>
</chassis>
```

The method attribute can be set to "xml", "html", or "text" and it determines the format for the output file. The default method is "xml"; unless the root element of the output file is <html>, in which case the default method is "html". When method is set to "text", all of the text content is output, without any escaping or extra indenting.

Code example:

```
var $content = {
    <html> {
        <head> {
            <title> "Example HTML";
        }
        <body> {
            expr "First Line";
            <br>;
            expr "Second Line";
        }
    }
}
<saxon:output href="/var/tmp/xml-method" method="xml" indent="yes"> {
    copy-of $content;
}
<saxon:output href="/var/tmp/html-method" method="html"> {
    copy-of $content;
}
<saxon:output href="/var/tmp/text-method" method="text"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/xml-method
<?xml version="1.0"?>
<html>
  <head>
    <title>Example HTML</title>
  </head>
  <body>First Line<br/>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/html-method
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Example HTML</title>
</head>
<body>First Line<br>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/text-method
Example HTMLFirst LineSecond Line
```

By default, the **<saxon:output>** element overwrites the output file if it already exists; however, if the append attribute, which was added in Junos 11.1, is set to either "yes" or "true", then the new data will be appended to the file, leaving the existing file contents in place. If append is set to "yes" or "true" when writing using the "xml" method, then the XML declaration is never added, even if the file is being created; however, at the time of this writing, if append is included and is not set to "yes" or "true" then the XML declaration is always added, even if omit-xml-declaration is set to "yes", so it is best to only include the append attribute if it is being enabled, especially since it is off by default.

Code example:

```
<saxon:output href="/var/tmp/time-record" method="text"> {
    expr date:date-time() _ "\n";
}
<saxon:output href="/var/tmp/time-record" method="text" append="yes"> {
    expr date:date-time() _ "\n";
}
```

Output file:

```
2011-04-23T21:11:31Z
2011-04-23T21:11:31Z
```

A large caveat exists when using **<saxon:output>**: it always accesses the file system as user "nobody". This means that files cannot be overwritten or appended unless their permissions give write access to user "nobody" or to everyone, and files cannot be created unless the directory provides similar permissions. In addition, when the **<saxon:output>** element creates an output file, it is owned by user "nobody" and its file permissions are 644, meaning that while anyone can read it, only user "nobody" can edit or delete it. (As of Junos 10.0R3, super-users can delete these files from the CLI). As a workaround, create the file initially through the <file-put> RPC with permissions 666, giving everyone read and write access, and then have **<saxon:output>** overwrite the original file. As a result, the file will be owned by the executing script's user and will have the desired contents as created by **<saxon:output>**.

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
```

```
            }
        }
    }
    <saxon:output href="/var/tmp/destination-config" indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-r--r--  1 nobody wheel       405 Apr 23 22:01 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

Example code:

```
    var $rpc = {
        <get-configuration> {
            <configuration> {
                <event-options> {
                    <destinations>;
                }
            }
        }
    }
    var $filename = "/var/tmp/destination-config";
    var $put-rpc = {
        <file-put> {
            <filename> $filename;
            <permission> "666";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "To be overwritten";
        }
    }
    var $results = jcs:invoke( $put-rpc );
    <saxon:output href=$filename indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-rw-rw-  1 jnpr  staff       405 Apr 23 22:07 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
```

```
            <name>local</name>
            <archive-sites>
                <name>/var/tmp</name>
            </archive-sites>
        </destinations>
    </event-options>
</configuration>
```

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in the output file:

Code example:

```
<saxon:output href="/var/tmp/output" doctype-system="local.dtd"> {
    <parent> {
        <child>;
        <child>;
        <child>;
    }
}
```

Output file:

```
<?xml version="1.0"?>
<!DOCTYPE parent SYSTEM "local.dtd">
<parent><child/><child/><child/></parent>
```

The encoding attribute determines what character encoding format should be used when writing the output file. The following (case-insensitive) values can be used: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii. The default value is utf-8.

Code example:

```
<saxon:output href="/var/tmp/output-html" encoding="html"> {
    <copyright> "©";
}
```

Output file:

```
<?xml version="1.0" encoding="html"?>
<copyright>&copy;</copyright>
```

The indent attribute can be set to either “yes” or “no”, and it determines whether indentation should be performed or not. It is off by default for the “xml” method and on by default for the “html” method. It has no effect on text files.

Code example:

```
var $content = {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
        }
    }
}
<saxon:output href="/var/tmp/output1" method="xml"> {
    copy-of $content;
}
<saxon:output href="/var/tmp/output2" method="xml" indent="yes"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/output1
<?xml version="1.0"?>
<interfaces><interface><name>ge-0/0/0</name></interface></interfaces>

jnpr@srx210> file show /var/tmp/output2
<?xml version="1.0"?>
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
  </interface>
</interfaces>
```

A XML declaration is included by default when using the "xml" method (unless appending is enabled). To disable the XML declaration, set the omit-xml-declaration attribute to "yes". This attribute has no effect when using the "html" or "text" methods.

Code example:

```
<saxon:output href="/var/tmp/output" indent="yes"
               omit-xml-declaration="yes"> {
    <details> {
        <script> $script;
        <user> $user;
    }
}
```

Output file:

```
<details>
  <script>test.slax</script>
  <user>jnpr</user>
</details>
```

By default, no standalone document declaration is included in the XML declaration, but if the standalone attribute is set to either "yes" or "no", then standalone is included in the declaration with the specified value.

Code example:

```
<saxon:output href="/var/tmp/context" indent="yes" standalone="yes" > {
    copy-of $junos-context/user-context;
}
```

Output file:

```
<?xml version="1.0" standalone="yes"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

The version attribute is used to alter the XML version value that is written in the XML declaration of the output file when the "xml" method is used. It should be set to "1.0", but changing it to other values does not alter the format of the document in any way.

The **<saxon:output>** element has an additional attribute: cdata-section-elements, which is intended to indicate what nodes in the XML output file should have their text contents enclosed within CDATA sections, but it is non-functional at the time of this writing. Also, the media-type attribute is not implemented and has no effect, and none of the SAXON specific attributes (such as saxon:indent-spaces) are implemented, either.

If an error occurs due to an inability to write to the output file, an incorrect attribute setting, or for some other reason, then the script fails and an error message is displayed to the script user for op scripts and commit scripts or to the event script output file. The script itself is unable to react to the error.

The Junos API contains a RPC element for writing to the disk: <file-put>, which has some capabilities that **<saxon:output>** does not have, including the ability to write in base64 format, to select file permissions, and to write to the non-local routing-engines. In addition, unlike **<saxon:output>**, the <file-put> RPC accesses the file system using the credentials of the user that is executing the script. However, <file-put> is incapable of writing in XML or HTML format or of appending to an existing file, so when a script needs to perform one of those three operations then **<saxon:output>** would be an appropriate element to use.

Junos contains support for five instruction elements that have equivalent functionality: **<exsl:document>**, **<redirect:write>**, **<saxon:output>**, **<xsl:document>**, and **<xt:document>. All attributes and functionality are common among the five elements except for some variances in support for the different file selector attributes: file, href, and select. Given that all five elements perform the same operation, it is recommended that the <xsl:document> element be used in place of the others because, unlike the other four, it does not need to have an extension namespace defined.**

Example

This op script demonstrates how to use the five file writing instruction elements.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl extension = "http://exslt.org/common";
ns saxon extension = "http://icl.com/saxon";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
ns xt extension = "http://www.jclark.com/xt";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* <redirect:write> <xsl:document> <saxon:output> and appending */
        var $append-file = "/var/tmp/append-example";
        <redirect:write href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <xsl:document href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <saxon:output href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }

        /* <exsl:document> and indenting */
        <exsl:document href="/var/tmp/indent-example" indent="yes"> {
            <chassis> {
                <platform> "m40e";
                <platform> "mx960";
                <platform> "ex8216";
            }
        }

        /* <xt:document> and text output */
        <xt:document href="/var/tmp/text-example" method="text"> {
            expr "Class-name: " _ $junos-context/user-context/class-name _ "\n";
```

```
            expr "User-name: " _ $junos-context/user-context/user _ "\n";
        }
    }
}
```

Output

```
jnpr@srx210> file list detail /var/tmp/*-example
-rw-r--r--  1 nobody wheel       318 Apr 25 15:31 /var/tmp/append-example
-rw-r--r--  1 nobody wheel       130 Apr 25 15:31 /var/tmp/indent-example
-rw-r--r--  1 nobody wheel        41 Apr 25 15:31 /var/tmp/text-example
total 3

jnpr@srx210> file show /var/tmp/append-example
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
<localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>

jnpr@srx210> file show /var/tmp/indent-example
<?xml version="1.0"?>
<chassis>
  <platform>m40e</platform>
  <platform>mx960</platform>
  <platform>ex8216</platform>
</chassis>

jnpr@srx210> file show /var/tmp/text-example
Class-name: j-super-user
User-name: jnpr
```

<xsl:apply-imports>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:apply-imports>;

Description

Match templates defined within a script override any similar match templates found within the script's import files; however, the **<xsl:apply-imports>** instruction element can be used to invoke the overridden template from the import file, but this element is not commonly used because named templates are more common in Junos SLAX scripts than in match templates.

If the current match template was invoked with a particular mode, then that mode will be used to select the imported template as well. It is an error to use **<xsl:apply-imports>** within a **for-each** loop.

Example

The junos.xsl import file contains a “match /” template that is used by commit scripts to automatically enclose their results in a result tree element, but the default “match /” template has no effect on op scripts because the “match /” template they contain always overrides the imported “match /” template. This op script uses the **<xsl:apply-imports>** element to invoke that overridden template from the junos.xsl import file so that the calling template can capture and display the result tree element that is added to commit scripts by default.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

    /*
     * Capture the result tree document element that junos.xsl adds
     * automatically to commit scripts.
     */
    var $capture := {
        /* Invokes the match / template in the import file */
        <xsl:apply-imports>;
    }
    /* Display the name */
    expr jcs:output( "Element added automatically by junos.xsl: <", name( $capture/* ), ">" );
}
```

Output

```
Element added automatically by junos.xsl: <commit-script-results>
```

<xsl:attribute>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:attribute name=*“name”* namespace=*“URI”***>** *“value”***;**

Description

The **<xsl:attribute>** instruction element adds a XML attribute to an element that is being created for the result tree or for variable assignment. It is typically used in conjunction with the **<xsl:element>** instruction element; however, it can also be used when an element is created through its inclusion within the script and might be, in

particular, necessary if the attribute's name cannot be coded into the script, as shown below:

Code:

```
<example-element> {
    <xsl:attribute name=$new-name> "value";
}
```

Created element & attribute (if $new-name is "type"):

```
<example-element type="value">
```

Example of **<xsl:element>** and **<xsl:attribute>** being used together:

Code:

```
<xsl:element name="another-example"> {
    <xsl:attribute name="another-attribute"> "another-value";
}
```

Created element & attribute:

```
<another-example another-attribute="another-value">
```

The **<xsl:attribute>** instruction element is also used within the **<xsl:attribute-set>** element to define the attributes that constitute the attribute-set. Attributes added to an element through **<xsl:attribute>** are added after those added through any attribute-sets, as well as any included explicitly within the element's tag. If more than one attribute with the same name is assigned to an element then the latter attribute overrides the prior value.

The name attribute of **<xsl:attribute>** is mandatory and must be a qualified name (see Appendix B), which can optionally contain a namespace prefix, but if a namespace prefix is included for the name then the prefix must be defined within the script or else the namespace attribute must also be included. The namespace attribute of **<xsl:attribute>** is optional and indicates what URI should be associated with the namespace prefix when one is present, or if a namespace prefix is not part of the attribute name, then one is created automatically. The values of both the name and namespace attributes are treated as attribute value templates, so they can be set to a hardcoded string, variable, or XPath expression.

The **<xsl:attribute>** element's value must be a string, which can be assigned directly, as shown above, or it can be assigned by enclosing additional code within curly brackets:

Code:

```
<chassis> {
    <xsl:attribute name="type"> {
        if(  $physical ) {
            expr "physical";
        }
        else {
            expr "virtual";
        }
    }
}
```

Created element & attribute (if $physical is true):

```
<chassis type="physical">
```

The **<xsl:attribute>** instruction element must be applied to an element prior to the assignment of any child elements or text content; otherwise, it will cause an error. It is not possible to define an attribute that has a prefix of xmlns.

Example

This op script demonstrates the use of **<xsl:attribute>** in conjunction with **<xsl:element>**.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display protocol options */
        var $config-rpc = {
            <get-configuration inherit="inherit"> {
                <configuration> {
                    <protocols>;
                }
            }
        }
        var $config = jcs:invoke( $config-rpc );
        expr jcs:output( "Active protocols: " );
        for-each( $config/protocols/* ) {
            expr jcs:output( name() );
        }

        /* Select protocol to deactivate */
        var $protocol = jcs:get-input( "Select protocol to deactivate: " );
        var $attribute = "inactive";

        /* Build configuration change */
        var $configuration = {
            <configuration> {
                <protocols> {
                    <xsl:element name=$protocol> {
                        <xsl:attribute name=$attribute> $attribute;
                    }
                }
            }
        }

        /* Apply it and return any errors/warnings */
        var $connection = jcs:open();
        call jcs:load-configuration( $connection, $configuration);
    }
}
```

Output

```
Active protocols:
bgp
isis
ospf
rip
Select protocol to deactivate: rip
```

```
jnpr@srx210> show configuration protocols | match inactive
inactive: rip {
```

<xsl:attribute-set>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:attribute-set name=*"name"* use-attribute-sets=*"names"***>** {

  <xsl:attribute> ...
}

Description

The **<xsl:attribute-set>** top-level element creates a set of attributes which can be applied when creating or copying elements. The name attribute is required and is a qualified name (see Appendix B). The use-attribute-sets attribute is optional, but can be set to a whitespace (space, tab, etc.) separated string of attribute-set names whose attributes are added into the current set. Both the name and the use-attribute-sets attributes must be assigned a hardcoded string value.

The attributes that make up an attribute-set are defined by including **<xsl:attribute>** instruction elements as child elements of **<xsl:attribute-set>** in this manner:

```
<xsl:attribute-set name="mx960"> {
    <xsl:attribute name="type"> "mx960";
    <xsl:attribute name="slots"> "12";
}
```

If the same attribute name is added to the set through another attribute-set as well as directly through a **<xsl:attribute>** instruction element then the **<xsl:attribute>** overrides the other value.

Attribute-sets can be applied to the **<xsl:attribute-set>**, **<xsl:element>**, and **<xsl:copy>** elements through the use-attribute-sets attribute. They can also be applied to an element that is included within the script code through the xsl:use-attribute-sets attribute.

Code:

```
<xsl:element name="chassis" use-attribute-sets="mx960">;
```

Created Element & Attribute:

```
<chassis type="mx960" slots="12">
```

Code:

```
<chassis xsl:use-attribute-sets="mx960">;
```

Created Element & Attribute:

```
<chassis type="mx960" slots="12">
```

Attribute-sets that have the same name are merged together into a single attribute-set.

Example

This op script demonstrates how to use **<xsl:attribute-set>** to assign multiple attributes to an element.

Code

```
    version 1.0;

    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    /* Get current last order */
    var $config = jcs:invoke( "get-configuration" );
    var $last-authentication-order = $config/system/authentication-order[last()];

    /* Create the attribute-set */
    <xsl:attribute-set name="insert-after"> {
        <xsl:attribute name="insert"> "after";
        <xsl:attribute name="name"> $last-authentication-order;
    }

    match / {
        <op-script-results> {

            /* Show current order */
            expr jcs:output( "Current authentication order: " );
            for-each( $config/system/authentication-order ) {
                expr jcs:output( . );
            }

            /* Gather the information */
            var $item-to-move = jcs:get-input( "Move to end: " );

            /* Build configuration change */
            var $configuration = {
                <configuration> {
                    <system> {
                        <authentication-order xsl:use-attribute-sets="insert-after"> $item-to-
move;
                    }
                }
            }

            /* Apply it and return any errors/warnings */
            var $connection = jcs:open();
            call jcs:load-configuration( $connection, $configuration);
        }
    }
```

Output

```
    Current authentication order:
    password
    radius
    tacplus
    Move to end: radius

    jnpr@srx210> show configuration system authentication-order
```

```
    authentication-order [ password tacplus radius ];
```

<xsl:copy>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:copy use-attribute-sets="*names*"**>;**

Description

The **<xsl:copy>** instruction element performs a shallow-copy of the current node into the result tree (which could be redirected into a result tree fragment variable). The shallow-copy behavior of **<xsl:copy>** is in contrast to the more commonly used deep-copy performed by the **copy-of** SLAX statement in that only the current node and its namespace nodes are copied, while attributes, child elements, and the text content of element nodes are not. Also, there is no way to select a node other than the current node, so **<xsl:copy>** is mainly used within **for-each** loops or templates where the desired node is already set as the current node.

The use-attribute-sets attribute is optional but can be set to a whitespace (space, tab, etc.) separated string of attribute-set names, causing each of the attributes within the attribute-sets to be added to the newly copied element. The attribute's value must be set as a hardcoded string.

If the **<xsl:copy>** element has child elements or text content, and the current node being copied is an element or root node, then these are added to the newly created element :

Code:

```
    for-each( $configuration//interface ) {
        <xsl:copy> {
            <xsl:attribute name="active"> "active";
            <name> name;
        }
    }
```

Copied element (if the name child node is "ge-0/0/0"):

```
    <interface active="active"> {
        <name> "ge-0/0/0";
    }
```

Example

This op script demonstrates the difference between the shallow-copy performed by **<xsl:copy>** and the deep-copy performed by the **copy-of** SLAX statement.

Code

```
    version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $interfaces := {
            <interfaces> {
                <apply-groups> "jumbo-mtu";
                <interface> {
                    <name> "ge-0/0/0";
                }
                <interface disable="disable"> {
                    <name> "ge-1/0/0";
                }
            }
        }

        /* Create a shallow-copy of the interfaces child nodes */
        var $shallow-copy := {
            for-each( $interfaces/interfaces/* ) {
                <xsl:copy>;
            }
        }

        /* Create a deep-copy of the interfaces child nodes */
        var $deep-copy := {
            for-each( $interfaces/interfaces/* ) {
                copy-of .;
            }
        }

        /* Highlight the differences */
        <output> "Shallow-copy:";
        for-each( $shallow-copy/* ) {
            call output-xml-node();
        }
        <output> str:padding( 8, "_" ) _ "\nDeep-copy:";
        for-each( $deep-copy/* ) {
            call output-xml-node();
        }
    }
}

/* Output the element and its attributes and text content */
template output-xml-node( $indent = 0 ) {

    /* No text */
    if( jcs:empty( text() ) ) {
        <output> str:padding( $indent ) _ "Element: " _ name();
    }
    /* Display text */
    else {
        <output> str:padding( $indent ) _ "Element: " _ name() _ " = " _ .;
    }

    /* Include any attributes */
    for-each( @* ) {
        <output> str:padding( $indent + 4 ) _ "Attribute: " _ name() _ " = " _ .;
```

```
        }

        /* Recurse if children are present */
        for-each( * ) {
            call output-xml-node( $indent = $indent + 4 );
        }
    }
```

Output

```
Shallow-copy:
Element: apply-groups
Element: interface
Element: interface
________
Deep-copy:
Element: apply-groups = jumbo-mtu
Element: interface
    Element: name = ge-0/0/0
Element: interface
    Attribute: disable = disable
    Element: name = ge-1/0/0
```

<xsl:decimal-format>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:decimal-format decimal-separator="." digit="#" grouping-separator="," infinity="*Infinity*"

minus-sign="-" name="*name*" NaN="*NaN*" pattern-separator=";" percent="%"

per-mille="‰" zero-digit="*0*"**>;**

Description

The **<xsl:decimal-format>** top-level element is used to create a decimal-format, which controls the format of the **format-number**() function's pattern string as well as the appearance of its formatted number.

As a top-level element, the **<xsl:decimal-format>** element must be defined outside of any templates or functions. If no name attribute is included then the **<xsl:decimal-format>** definition becomes the default decimal-format used by the **format-number**() function. It is an error to create more than one decimal-format with the same name.

All of the attributes of **<xsl:decimal-format>** are optional, but when included they must be set to a hardcoded string value. Any attributes that are not included within a decimal-format definition will use their default value instead.

Attribute Name	Default Value	Description
decimal-separator	.	Separates the number into integer and decimal
digit	#	Represents an optional digit
grouping-separator	,	Separates the number into groups such as thousands, etc.
infinity	Infinity	String used to represent infinity
minus-sign	-	Character used as the minus sign
name		Name of the decimal-format (should be a valid qualified name – see Appendix B)
NaN	NaN	String used to represent not-a-number
pattern-separator	;	Separates the positive number and negative number patterns
per-mille	‰	Character used as the per-mille sign
percent	%	Character used as the percent sign
zero-digit	0	Represents a required digit as well as the character base of the digits in the formatted number

See the **format-number**() function description for details on how these different values effect the pattern string as well as the formatted number.

Example

This op script demonstrates how to use decimal-formats created by the **<xsl:decimal-format>** element.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Alternate decimal format */
<xsl:decimal-format name="alternate" grouping-separator="." decimal-separator=",">;

/* Alternate names */
<xsl:decimal-format name="alternate-names" infinity="very-large" NaN="not-a-number">;

match / {
    <op-script-results> {

        var $num1 = 123456.789;
```

```
            /* Default separation */
            <output> format-number( $num1, "###,###.###" );

            /* Alternate grouping separator */
            <output> format-number( $num1, "###.###,###", "alternate" );

            var $NaN = "ABC";
            var $infinity = 100 div 0;

            /* Default NaN and Infinity */
            <output> format-number( $NaN, "#" );
            <output> format-number( $infinity, "#" );

            /* Alternate NaN and Infinity */
            <output> format-number( $NaN, "#", "alternate-names" );
            <output> format-number( $infinity, "#", "alternate-names" );
        }
    }
```

Output

```
123,456.789
123.456,789
NaN
Infinity
not-a-number
very-large
```

<xsl:document>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:document append="*yes|no*" doctype-public="*string*" doctype-system="*string*"

encoding="*format*" href="*filename*" indent="*yes|no*" method="*html|text|xml*"

omit-xml-declaration="*yes|no*" standalone="*yes|no*" version="*string*" **>** {

File content...

}

Description

The **<xsl:document>** instruction element writes data to an output file on the local disk drive in XML, HTML, or text format. The filename is indicated through the required href attribute, which like all the other attributes of **<xsl:document>** is handled as an attribute value template, so its value can be specified as a hardcoded string, a variable, or a XPath expression.

It is recommended that the full path to the output file be provided, because while the default directory for op

scripts and commit scripts is /var/tmp, the default directory at times for event scripts is the root directory /, which cannot be written to by **<xsl:document>**. The only allowed URI scheme for the filename is "file", and it can be included as part of the href attribute value, but is not required and there is no difference in behavior if it is not included (e.g. href="file:///var/tmp/output").

The **<xsl:document>** element can only write to the local disk. Writing via networking protocols such as FTP or HTTP is not supported, and neither is writing to other routing-engines. Also, the full path specified in the href attribute must already exist because the script processor in Junos does not create new directories as part of the **<xsl:document>** operation.

All contents of the **<xsl:document>** element are written to the output file:

Code example:

```
<xsl:document href="/var/tmp/example-file" indent="yes"> {
    <chassis> {
        <name> "M40e";
        <name> "EX8216";
    }
}
```

Output file:

```
<?xml version="1.0"?>
<chassis>
  <name>M40e</name>
  <name>EX8216</name>
</chassis>
```

The method attribute can be set to "xml", "html", or "text" and it determines the format for the output file. The default method is "xml"; unless the root element of the output file is <html>, in which case the default method is "html". When method is set to "text", all of the text content is output, without any escaping or extra indenting.

Code example:

```
var $content = {
    <html> {
        <head> {
            <title> "Example HTML";
        }
        <body> {
            expr "First Line";
            <br>;
            expr "Second Line";
        }
    }
}
<xsl:document href="/var/tmp/xml-method" method="xml" indent="yes"> {
    copy-of $content;
}
<xsl:document href="/var/tmp/html-method" method="html"> {
    copy-of $content;
}
<xsl:document href="/var/tmp/text-method" method="text"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/xml-method
<?xml version="1.0"?>
<html>
  <head>
```

```
    <title>Example HTML</title>
  </head>
  <body>First Line<br/>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/html-method
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Example HTML</title>
</head>
<body>First Line<br>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/text-method
Example HTMLFirst LineSecond Line
```

By default, the **<xsl:document>** element overwrites the output file if it already exists; however, if the append attribute, which was added in Junos 11.1, is set to either "yes" or "true", then the new data will be appended to the file, leaving the existing file contents in place. If append is set to "yes" or "true" when writing using the "xml" method, then the XML declaration is never added, even if the file is being created; however, at the time of this writing, if append is included and is not set to "yes" or "true" then the XML declaration is always added, even if omit-xml-declaration is set to "yes", so it is best to only include the append attribute if it is being enabled, especially since it is off by default.

Code example:

```
<xsl:document href="/var/tmp/time-record" method="text"> {
    expr date:date-time() _ "\n";
}
<xsl:document href="/var/tmp/time-record" method="text" append="yes"> {
    expr date:date-time() _ "\n";
}
```

Output file:

```
2011-04-23T21:11:31Z
2011-04-23T21:11:31Z
```

A large caveat exists when using **<xsl:document>**; it always accesses the file system as user "nobody". This means that files cannot be overwritten or appended unless their permissions give write access to user "nobody" or to everyone, and files cannot be created unless the directory provides similar permissions. In addition, when the **<xsl:document>** element creates an output file, it is owned by user "nobody" and its file permissions are 644, meaning that while anyone can read it, only user "nobody" can edit or delete it. (As of Junos 10.0R3, super-users can delete these files from the CLI). As a workaround, create the file initially through the <file-put> RPC with permissions 666, giving everyone read and write access, and then have **<xsl:document>** overwrite the original file. As a result, the file will be owned by the executing script's user and will have the desired contents as created by **<xsl:document>**.

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
        }
    }
}
<xsl:document href="/var/tmp/destination-config" indent="yes"> {
    copy-of jcs:invoke( $rpc );
```

```
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-r--r--  1 nobody wheel       405 Apr 23 22:01 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

Example code:

```
    var $rpc = {
        <get-configuration> {
            <configuration> {
                <event-options> {
                    <destinations>;
                }
            }
        }
    }
    var $filename = "/var/tmp/destination-config";
    var $put-rpc = {
        <file-put> {
            <filename> $filename;
            <permission> "666";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "To be overwritten";
        }
    }
    var $results = jcs:invoke( $put-rpc );
    <xsl:document href=$filename indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-rw-rw-  1 jnpr  staff        405 Apr 23 22:07 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
```

```
    </event-options>
</configuration>
```

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in the output file:

Code example:

```
<xsl:document href="/var/tmp/output" doctype-system="local.dtd"> {
    <parent> {
        <child>;
        <child>;
        <child>;
    }
}
```

Output file:

```
<?xml version="1.0"?>
<!DOCTYPE parent SYSTEM "local.dtd">
<parent><child/><child/><child/></parent>
```

The encoding attribute determines what character encoding format should be used when writing the output file. The following (case-insensitive) values can be used: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii. The default value is utf-8.

Code example:

```
<xsl:document href="/var/tmp/output-html" encoding="html"> {
    <copyright> "©";
}
```

Output file:

```
<?xml version="1.0" encoding="html"?>
<copyright>&copy;</copyright>
```

The indent attribute can be set to either “yes” or “no”, and it determines whether indentation should be performed or not. It is off by default for the “xml” method and on by default for the “html” method. It has no effect on text files.

Code example:

```
var $content = {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
        }
    }
}
<xsl:document href="/var/tmp/output1" method="xml"> {
    copy-of $content;
}
<xsl:document href="/var/tmp/output2" method="xml" indent="yes"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/output1
<?xml version="1.0"?>
<interfaces><interface><name>ge-0/0/0</name></interface></interfaces>

jnpr@srx210> file show /var/tmp/output2
```

```
<?xml version="1.0"?>
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
  </interface>
</interfaces>
```

A XML declaration is included by default when using the “xml” method (unless appending is enabled). To disable the XML declaration, set the omit-xml-declaration attribute to “yes”. This attribute has no effect when using the “html” or “text” methods.

Code example:

```
<xsl:document href="/var/tmp/output" indent="yes" omit-xml-declaration="yes"> {
    <details> {
        <script> $script;
        <user> $user;
    }
}
```

Output file:

```
<details>
  <script>test.slax</script>
  <user>jnpr</user>
</details>
```

By default, no standalone document declaration is included in the XML declaration, but if the standalone attribute is set to either “yes” or “no”, then standalone is included in the declaration with the specified value.

Code example:

```
<xsl:document href="/var/tmp/context" indent="yes" standalone="yes" > {
    copy-of $junos-context/user-context;
}
```

Output file:

```
<?xml version="1.0" standalone="yes"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

The version attribute is used to alter the XML version value that is written in the XML declaration of the output file when the “xml” method is used. It should be set to “1.0”, but changing it to other values does not alter the format of the document in any way.

The **<xsl:document>** element has an additional attribute: cdata-section-elements, which is intended to indicate what nodes in the XML output file should have their text contents enclosed within CDATA sections, but it is non-functional at the time of this writing. Also, the media-type attribute is not implemented and has no effect.

If an error occurs due to an inability to write to the output file, an incorrect attribute setting, or for some other reason, then the script fails and an error message is displayed to the script user for op scripts and commit scripts or to the event script output file. The script itself is unable to react to the error.

The Junos API contains a RPC element for writing to the disk: **<file-put>**, which has some capabilities that **<xsl:document>** does not have, including the ability to write in base64 format, to select file permissions, and to write to the non-local routing-engines. In addition, unlike **<xsl:document>**, the **<file-put>** RPC accesses the file system using the credentials of the user that is executing the script. However, **<file-put>** is incapable of writing in XML or HTML format or of appending to an existing file, so when a script needs to perform one of those

three operations then **<xsl:document>** would be an appropriate element to use.

Junos contains support for five instruction elements that have equivalent functionality: **<exsl:document>**, **<redirect:write>**, **<saxon:output>**, **<xsl:document>**, and **<xt:document>.** All attributes and functionality are common among the five elements except for some variances in support for the different file selector attributes: file, href, and select. Given that all five elements perform the same operation, it is recommended that the **<xsl:document>** element be used in place of the others because, unlike the other four, it does not need to have an extension namespace defined.

Example

This op script demonstrates how to use the five file writing instruction elements.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl extension = "http://exslt.org/common";
ns saxon extension = "http://icl.com/saxon";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
ns xt extension = "http://www.jclark.com/xt";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* <redirect:write> <xsl:document> <saxon:output> and appending */
        var $append-file = "/var/tmp/append-example";
        <redirect:write href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <xsl:document href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <saxon:output href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }

        /* <exsl:document> and indenting */
        <exsl:document href="/var/tmp/indent-example" indent="yes"> {
            <chassis> {
                <platform> "m40e";
                <platform> "mx960";
                <platform> "ex8216";
            }
        }

        /* <xt:document> and text output */
        <xt:document href="/var/tmp/text-example" method="text"> {
            expr "Class-name: " _ $junos-context/user-context/class-name _ "\n";
            expr "User-name: " _ $junos-context/user-context/user _ "\n";
        }
    }
}
```

Output

```
    jnpr@srx210> file list detail /var/tmp/*-example
    -rw-r--r--  1 nobody wheel       318 Apr 25 15:31 /var/tmp/append-example
    -rw-r--r--  1 nobody wheel       130 Apr 25 15:31 /var/tmp/indent-example
    -rw-r--r--  1 nobody wheel        41 Apr 25 15:31 /var/tmp/text-example
    total 3

    jnpr@srx210> file show /var/tmp/append-example
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>

    jnpr@srx210> file show /var/tmp/indent-example
    <?xml version="1.0"?>
    <chassis>
      <platform>m40e</platform>
      <platform>mx960</platform>
      <platform>ex8216</platform>
    </chassis>

    jnpr@srx210> file show /var/tmp/text-example
    Class-name: j-super-user
    User-name: jnpr
```

<xsl:element>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:element name=*"name"* namespace=*"URI"* use-attribute-sets=*"names"* **>** *"value"***;**

Description

The **<xsl:element>** instruction element creates a XML element for the result tree or for variable assignment. It is typically used when the element's name cannot be coded into the script, because if the name is already known then it would be possible to code the element directly into the script instead.

For example, the following code snippets create an identical result:

Code:

```
    <example-element> "value";
```

Code:

```
    <xsl:element name="example-element"> "value";
```

Created element:

```
<example-element> "value"
```

The name attribute of **<xsl:element>** is mandatory and must be a qualified name (see Appendix B), which can optionally contain a namespace prefix, but if a namespace prefix is included for the name then the prefix must be defined within the script or else the namespace attribute must also be included. The namespace attribute of **<xsl:element>** is optional and indicates what URI should be associated with the namespace prefix when one is present, or if a namespace prefix is not part of the element name then the namespace attribute's value becomes the default namespace for the element.

The use-attribute-sets attribute is optional, but can be set to a whitespace (space, tab, etc.) separated string of attribute-set names, causing each of the attributes within the attribute-sets to be added to the element. Attributes can also be added through the **<xsl:attribute>** instruction element, which must appear prior to any child elements or text content.

The values of both the name and namespace attributes are attribute value templates, so they can be set to a hardcoded string, a variable, or a XPath expression. According to the XSLT specification, the use-attribute-sets attribute should only be set to a hardcoded string value; however, at the time of this writing its value is treated as an attribute value template as well.

Elements can be assigned child elements or text content by following the standard SLAX XML syntax rules. Here are examples of how to provide content to elements created by **<xsl:element>**:

Assigning text content:

```
<xsl:element name="chassis"> "mx960";
```

Assigning child elements:

```
<xsl:element name="user"> {
    <name> "admin";
    <login-class> "super-user";
}
```

Multiple levels of **<xsl:elements>** instruction elements:

```
<xsl:element name="parent"> {
    <xsl:element name="child"> "child1";
    <xsl:element name="child"> "child2";
}
```

Example

This op script demonstrates the use of **<xsl:attribute>** in conjunction with **<xsl:element>**.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display protocol options */
        var $config-rpc = {
            <get-configuration inherit="inherit"> {
                <configuration> {
```

```
                    <protocols>;
                }
            }
        }
        var $config = jcs:invoke( $config-rpc );
        expr jcs:output( "Active protocols: " );
        for-each( $config/protocols/* ) {
            expr jcs:output( name() );
        }

        /* Select protocol to deactivate */
        var $protocol = jcs:get-input( "Select protocol to deactivate: " );
        var $attribute = "inactive";

        /* Build configuration change */
        var $configuration = {
            <configuration> {
                <protocols> {
                    <xsl:element name=$protocol> {
                        <xsl:attribute name=$attribute> $attribute;
                    }
                }
            }
        }

        /* Apply it and return any errors/warnings */
        var $connection = jcs:open();
        call jcs:load-configuration( $connection, $configuration);
    }
}
```

Output

```
Active protocols:
bgp
isis
ospf
rip
Select protocol to deactivate: rip

jnpr@srx210> show configuration protocols | match inactive
inactive: rip {
```

<xsl:fallback>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:fallback> {

```
    fallback code...
}
```

Description

The **<xsl:fallback>** instruction element is enclosed within another instruction element to indicate what fallback code should be run if the script processor does not recognize the enclosing instruction element, which could be an element within the xsl namespace or an element from an extension namespace. For example, because the <xsl:unknown-instruction> instruction element is unknown, the **<xsl:fallback>** code block is executed instead, writing a message to the syslog:

```
<xsl:unknown-instruction> {
    <xsl:fallback> {
        expr jcs:syslog( "daemon.error", "Instruction element is unknown!" );
    }
}
```

Because it is not possible to add new extension elements for Junos scripts (the allowed instruction elements are constant and well-defined) this instruction element is of limited value; however, the EXSLT extension elements were not supported prior to Junos 9.4, so a script might utilize the **<xsl:fallback>** instruction element to define what action should be taken when it is run in earlier versions that do not recognize the EXSLT extension elements it uses.

Note: At the time of this writing, result tree elements generated from the contents of a **<xsl:fallback>** instruction element are not inserted into the correct result tree hierarchy level when the enclosing instruction element is from an extension namespace. Instead, they are appended directly to the root node.

Example

This op script demonstrates how **<xsl:fallback>** can be used to react to the lack of support for an extension element.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns test extension = "test";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        /* Fake extension element */
        <test:fake> {
            expr jcs:output( "<test:fake> exists!" );

            <xsl:fallback> {
                expr jcs:output( "<test:fake> does not exist." );
            }
        }
    }
}
```

Output

```
<test:fake> does not exist.
```

<xsl:key>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:key match=*"nodes"* name=*"name"* use =*"value"* **>;**

Description

Keys can be used, as an alternative to IDs, to index the nodes within a XML document. They are defined by including **<xsl:key>** top-level elements within the script, which consist of a name, the nodes to index, and the value that is paired with the key name to reference the matching nodes. The **key**() function can then be used to retrieve the nodes that are referenced by a particular name and value.

The **<xsl:key>** top-level element has three mandatory attributes, all of which must be hardcoded as string values rather than being assigned through variables, parameters, or location paths:

- match – A pattern that matches the nodes for which keys should be created. Unlike a normal location path expression, there is no single context node that the pattern is based on; instead, a node is considered a match if it can be described by the pattern when the context is the node itself or any of its ancestors, meaning that a pattern of "name" would match a <name> node that is a child of <unit> as well as a <name> node that is a child of <interface>, etc.
- name – The key name. It should be a valid qualified name (see Appendix B).
- use – A location path expression used to assign the key's value. The use expression is evaluated for each key with the matching node as its context node.

The presence of **<xsl:key>** elements causes keys to be created in all XML documents, including the source tree as well as any XML content returned to the script through functions; however, the **key**() function works with the XML document of the current node, so the current node might have to be explicitly changed, by using a **for-each** loop, for example, in order to retrieve nodes from the desired XML document.

Example

This op script example indexes the XML results of the static route configuration according to the route's community and next-hop. It shows how to use the **key**() function to retrieve all nodes that are indexed according to a specific key name and value.

Code

```
version 1.0;
```

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

/* Define route keys */
<xsl:key name="community" match="route" use="community">;
<xsl:key name="next-hop" match="route" use="next-hop">;

match / {
    <op-script-results> {

        /* Retrieve the routing-options configuration */
        var $rpc = {
            <get-configuration> {
                <configuration> {
                    <routing-options> {
                        <static>;
                    }
                }
            }
        }
        var $routing-options = jcs:invoke( $rpc );

        /* Change current node to be within $routing-options so that key() will work */
        for-each( $routing-options ) {

            /* Grab the routes with a next-hop of 10.0.0.1 */
            <output> "Next-hop of 10.0.0.1:";
            for-each( key( "next-hop", "10.0.0.1" ) ) {
                <output> name;
            }

            /* Grab the routes with a community of no-export */
            <output> "Community of no-export:";
            for-each( key( "community", "no-export" ) ) {
                <output> name;
            }
        }
    }
}
```

jcs:invoke() results:

```
<configuration>
    <routing-options>
        <static>
            <route>
                <name>0.0.0.0/0</name>
                <next-hop>10.0.0.1</next-hop>
                <community>no-advertise</community>
            </route>
            <route>
                <name>192.168.0.0/16</name>
                <next-hop>10.0.0.2</next-hop>
                <community>no-export</community>
            </route>
            <route>
                <name>172.16.0.0/12</name>
                <next-hop>10.0.0.1</next-hop>
                <community>no-export</community>
```

```
            </route>
        </static>
    </routing-options>
</configuration>
```

Output

```
Next-hop of 10.0.0.1:
0.0.0.0/0
172.16.0.0/12
Community of no-export:
192.168.0.0/16
172.16.0.0/12
```

<xsl:message>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:message terminate=*"no"***>** *"message"***;**

Description

The **<xsl:message>** instruction element displays an error message and optionally exits the script. The terminate attribute is optional, but when set to "yes" it instructs the script to terminate immediately; however, this attribute must be set with a hardcoded "yes" or "no" value rather than a variable.

The text that should be displayed is assigned to the **<xsl:message>** element:

Code:

```
<xsl:message> "Invalid name";
```

Displayed message:

```
error: Invalid name
```

In response to this element, Junos omits two <xnm:error> elements, the first of which contains the element's text message, and the latter of which is blank (and can be ignored). The effect depends on the script type:

- op script: The message text is displayed with "error: " prepended.
- commit script: The message text is displayed with "error: " prepended and the commit process terminates with two errors, so commit scripts should only use **<xsl:message>** in response to actual errors that should halt the commit.
- event script: If an output file is configured then the message is captured in one of two ways, depending on the output format:
  - With text output, the message text is written to the file with "error: " prepended.
  - With xml output, the two <xnm:error> elements are written to the file.

If a script is called by another script through the <op-script> Junos API element, and the called script uses the **<xsl:message>** instruction element, then the script will return the two generated <xnm:error> elements to the calling script as part of its XML output.

Example

This op script demonstrates how to use the **<xsl:message>** element to both display an error message and to terminate the script early.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

param $number;

match / {
    <op-script-results> {

        if( not( $number ) ) {
            <xsl:message terminate="yes"> "You must specify a number!";
        }

        <output> $number _ " times 3 = " _ $number * 3;
    }
}
```

Output

```
jnpr@srx210> op xsl_message number 2
2 times 3 = 6

jnpr@srx210> op xsl_message
error: You must specify a number!
```

<xsl:namespace-alias>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:namespace-alias stylesheet-prefix="*prefix*" result-prefix="*prefix*"**>;**

Description

The **<xsl:namespace-alias>** top-level element is used to translate a namespace URI within the script's code into a different namespace URI in its result tree. This is a rarely used element, as the result tree is processed directly by Junos so there is generally not a reason to use **<xsl:namespace-alias>** within SLAX scripts; however, one possible use case is if the SLAX script is writing XSLT code to disk through the **<xsl:document>** instruction element, or one of its related extension elements. In that scenario, it is necessary to prevent the script processor from trying to execute the embedded XSLT elements, which can be accomplished by assigning a different namespace URI to the embedded XSLT elements within the SLAX script, and then using the **<xsl:namespace-alias>** element to translate that namespace URI to the XSLT namespace URI in the output file.

The stylesheet-prefix and the result-prefix attribute, which are set as hardcoded strings, indicate the prefixes associated with the namespace URIs that should be translated from SLAX script to result tree. The stylesheet-prefix is the namespace prefix used within the SLAX script, and the result-prefix is the prefix associated with the namespace URI, which the stylesheet-prefix's namespace should be converted to. These namespaces must be defined within the SLAX script, except for the XSLT namespace, which is defined automatically as part of SLAX script processing with the prefix "xsl". Also, the namespace URIs associated with the stylesheet-prefix and result-tree prefix attributes must be distinct; otherwise, there would be no need to translate between the two of them.

At the time of this writing, the namespace prefix used in the SLAX script (the stylesheet-prefix) is the same namespace prefix used in the result tree. Only the namespace URI that the prefix references is changed.

The string "#default" can be used to set the stylesheet-prefix or the result-prefix to the default namespace; however, at the time of this writing, entering a stylesheet-prefix of "#default" is not handled correctly and should be avoided.

Example

This op script shows how **<xsl:namespace-alias>** can be used to write an embedded XSLT script to disk through a SLAX script.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
/* Temporary namespace prefix and alias */
ns embedxsl = "http://xml.juniper.net/temp";

import "../import/junos.xsl";

<xsl:namespace-alias stylesheet-prefix="embedxsl" result-prefix="xsl">;

match / {
    <op-script-results> {
        <xsl:document href="/var/tmp/script.xsl" method="xml" indent="yes"> {

            /* Assign fake junos xnm and jcs attributes to ensure that the namespaces
               are added to the file */
            <embedxsl:stylesheet junos:blank="" xnm:blank="" jcs:blank="" version="1.0"> {
                <embedxsl:template match="/"> {
                    <op-script-results> {
                        <embedxsl:value-of select="jcs:output('Hello World!')">;
```

```
                    }
                }
            }
        }
    }
```

Output

```
jnpr@srx210> op xsl_namespace-alias

jnpr@srx210> file show /var/tmp/script.xsl
<?xml version="1.0"?>
<embedxsl:stylesheet xmlns:embedxsl="http://www.w3.org/1999/XSL/Transform"
xmlns:junos="http://xml.juniper.net/junos/*/junos" xmlns:xnm="http://xml.juniper.net/xnm/1.1/
xnm" xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0" junos:blank="" xnm:blank=""
jcs:blank="" version="1.0">
  <embedxsl:template match="/">
    <op-script-results>
      <embedxsl:value-of select="jcs:output('Hello World!')"/>
    </op-script-results>
  </embedxsl:template>
</embedxsl:stylesheet>

jnpr@srx210> op url file://var/tmp/script.xsl
Hello World!
```

<xsl:number>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:number count=*"nodes"* format=*"number-format"* from=*"nodes"* grouping-separator=*","*

grouping-size=*"3"* level=*"any|multiple|single"* value=*"number"* **>;**

Description

The **<xsl:number>** instruction element generates a formatted number string, based on either a provided value or the position of one or more nodes within the current XML document. The formatted number string is written to the result tree by default but is generally redirected to a variable or output method.

<xsl:number> formats single numbers when its value attribute is set or its level attribute is set to "any" or "single", or it can format multiple numbers, separated by customizable strings, by setting the level attribute to "multiple" and not including a value attribute.

Code:

```
<xsl:number value="5000" format="(1)">;
```

Number string:

```
"(5000)"
```

Code:

```
<xsl:number level="multiple" count="top|middle|bottom" format="1.1.1">;
```

Number string:

```
"4.2.2"
```

The format attribute dictates how the number will be formatted. It is a string value that is an attribute value template so it can be set to a hardcoded string, a variable, or a XPath expression. If it is not included then the default format is "1". The format string can contain the following parts:

start string – Any non-alphanumeric characters that precede the first number token within the format string become the start string, which is prepended to the formatted number string.

number token – One or more number tokens can be included in the format string, indicating what numbering format to use for the included numbers; however, more than one number is only included in the formatted number string if the level attribute is set to "multiple":

1 – Use decimal format

The 1 can be preceded by a variable number of 0s, indicating the minimum length of the formatted number, with preceding zeros added to the formatted number if necessary.

A pattern of "01" results in "01", "02"..."09", "10", "11", etc.

A pattern of "001" results in "001", "002", ... "099", "100", "101", etc.

A – Use upper-case alphabetic numbering

a – Use lower-case alphabetic numbering

I – Use upper-case Roman numbering

i – Use lower-case Roman numbering

token separator – The non-alphanumeric characters that separate number tokens in the format string are included in the formatted number string between the computed numbers.

end string – Any non-alphanumeric characters that follow the last number token in the format string become the end string, which is appended to the end of the formatted number string.

Format examples:

- format="(001)"
  - "(010)", "(051)", "(100)", "(1050)"
- format="1.1"
  - "2.3", "3.1", "10.25"
- format="<A.1.1>"
  - "<C.1.5>", "<AJ.2.1>", "<BE.15.2>"
- format="I-i"
  - "II-i", "IV-iii"

The value attribute specifies the number that should be formatted. When value is specified; the level, count, and from attributes are ignored and only a single number is included in the formatted number string. The value is a hardcoded expression string, meaning that it can consist of a number, a variable, a location path, etc, so long as the XPath expression is included within the string itself. SLAX-specific operators such as _ and == are not supported within the expression string, and setting the value attribute to a negative number is also not supported.

Code:

```
<xsl:number value="100" format="-I-">;
```

Number string:

```
"-C-"
```

Code:

```
<xsl:number value="100 - 95" format="1.">;
```

Number string:

```
"5."
```

Code:

```
var $example-number = 1000;
<xsl:number value="$example-number" format="1">;
```

Number string:

```
"1000"
```

The grouping-size and grouping-separator attributes can be set to cause the formatted number to be split into multiple groups, according to the grouping-size (such as 3 for thousands) with the grouping-separator character added between groups. These attributes only effect numbers expressed in decimal format. Neither attribute is set by default, and they must both be set in order to take effect. Their values must be set as hard-coded strings.

Code:

```
<xsl:number value="123456" grouping-separator="," grouping-size="3" format="1">;
```

Number string:

```
"123,456"
```

Code:

```
<xsl:number value="9000" grouping-separator="." grouping-size="3" format="000001">;
```

Number string:

```
"009.000"
```

When the value attribute is not set, the count, level, and from attributes work together to determine what nodes should be counted, how they should be counted, and whether multiple levels of counting should occur or not.

The count attribute is a hardcoded string that contains the pattern describing which nodes should be counted. If count is not specified then it defaults to nodes with the same name as the current node.

Code:

```
var $data := {
    <parent> {
        <child>;
        <child>;
        <child>;
    }
```

```
}
for-each( $data//child ) {
    <output> {
        <xsl:number count="child">;
    }
}
```

Output:

```
1
2
3
```

The level attribute indicates what type of counting should be performed. It is set as a hardcoded string to one of three values:

- Single –Perform only one count. The node to be counted is chosen as either the current node, if it matches the count pattern, or the nearest ancestor that matches the count pattern. The position of the node in document order, relative to its siblings that also match the count parameter, is used as the number to be formatted. This is the default setting if the level attribute is not included.
- Multiple – Separately count all nodes that match the count pattern and are either the current node or an ancestor of the current node. The position of each node in document order, relative to its siblings that also match the count parameter, is used as one of the numbers to be formatted.
- Any – Perform only one count. The node to be counted is chosen as either the current node, if it matches the count pattern, or its nearest ancestor or preceding node that matches the count pattern. The position of the node in document order, relative to all other matching nodes that are ancestors or precede the node, is used as the number to be formatted.

Examples:

```
var $data := {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
            <unit> {
                <name> "0";
            }
        }
        <interface> {
            <name> "ge-0/0/1";
            <unit> {
                <name> "10";
            }
            <unit> {
                <name> "20";
            }
        }
    }
}
```

Code:

```
for-each( $data//interface ) {
    var $number = {
        <xsl:number count="interface" level="single" format="i">;
    }
    expr jcs:output( $number, ". ", name );
}
```

Output:

```
i. ge-0/0/0
ii. ge-0/0/1
```

Code:

```
for-each( $data//unit ) {
    var $number = {
        <xsl:number count="interfaces|interface|unit" level="multiple"
        format="(A.1.1)">;
    }
    expr jcs:output( $number, ". ", name );
}
```

Output:

```
(A.1.1). 0
(A.2.1). 10
(A.2.2). 20
```

Code:

```
for-each( $data//unit ) {
    var $number = { <xsl:number count="unit" level="any">; }
    expr jcs:output( $number, ". ", name );
}
```

Output:

```
1. 0
2. 10
3. 20
```

The from attribute is a hardcoded pattern string that is used to constrain the counting to only node descendants of the nearest ancestor that matches the from pattern when level is set to "single" or "multiple", or if the level is set to "any" then the counting is constrained to only nodes that follow the nearest ancestor or preceding node of the current node that matches the from pattern. In other words, it can be regarded as instruction to start counting "from" the indicated node, rather than always going as far back as possible.

Code:

```
var $data := {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
            <unit> {
                <name> "0";
            }
        }
        <interface> {
            <name> "ge-0/0/1";
            <unit> {
                <name> "10";
            }
            <unit> {
                <name> "20";
            }
        }
    }
}
for-each( $data//name ) {
    var $number = {
        <xsl:number count="name" from="interface" level="any" format="a">;
    }
    expr jcs:output( $number, ". ", . );
}
```

Output:

```
a. ge-0/0/0
b. 0
```

```
a. ge-0/0/1
b. 10
c. 20
```

Because the counting is based on the document position, not the sorted position, use of **<xsl:number>** in a **for-each** loop that has been sorted through the **<xsl:sort>** instruction element could result in unexpected numbering:

Code:

```
var $data := {
    <users> {
        <user> "jnpr";
        <user> "admin";
        <user> "read-only";
    }
}
for-each( $data//user ) {
    <xsl:sort>;
    var $number = { <xsl:number>; }
    expr jcs:output( $number, ". ", . );
}
```

Output:

```
2. admin
1. jnpr
3. read-only
```

The **<xsl:number>** element writes its result into the result tree, but this can be redirected into a variable through code similar to this:

```
var $number = { <xsl:number count="interface" format="A">; }
```

Alternatively, the formatted number string can be written directly to the console in op scripts by placing it as a child element of the <output> result tree element:

```
<output> {
    <xsl:number format="[i]">;
}
```

Note: The XSLT specification mentions two other attributes for **<xsl:number>**: lang and letter-value, but neither of these are supported.

Example

This op script demonstrates how to use the **<xsl:number>** element to generate multiple level numbers based on XML data.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        var $routing-options := {
```

```
            <routing-options> {
                <static> {
                    <route> {
                        <name> "0.0.0.0/0";
                        <next-hop> "10.0.0.1";
                    }
                    <route> {
                        <name> "192.168.0.0/16";
                        <next-hop> "10.0.0.2";
                        <next-hop> "10.0.0.1";
                        <next-hop> "10.0.0.3";
                    }
                    <route> {
                        <name> "172.16.0.0/12";
                        <next-hop> "10.0.0.1";
                        <next-hop> "10.0.0.2";
                    }
                }
            }
        }

        for-each( $routing-options//next-hop ) {
            var $number = {
                <xsl:number count="route | next-hop" level="multiple" format="1(01): ">;
            }
            <output> $number _ ../name _ "->" _ .;
        }
    }
}
```

Output

```
1(01): 0.0.0.0/0->10.0.0.1
2(01): 192.168.0.0/16->10.0.0.2
2(02): 192.168.0.0/16->10.0.0.1
2(03): 192.168.0.0/16->10.0.0.3
3(01): 172.16.0.0/12->10.0.0.1
3(02): 172.16.0.0/12->10.0.0.2
```

<xsl:output>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:output cdata-section-elements="*nodes*" doctype-public="*string*" doctype-system="*string*"

encoding="*format*" version="*string*"**>;**

Description

The **<xsl:output>** top-level element is intended to control how a XSLT processor outputs its result tree; however, Junos processes the result tree directly, rather than writing it to disk, so this element serves little purpose in Junos scripts besides to alter how the result tree document is copied into the trace file; but that effect is unlikely to be useful.

The **<xsl:output>** element has a number of attributes, but only the following have any effect: cdata-section-elements, doctype-public, doctype-system, encoding, and version. The other attributes of **<xsl:output>**, which are described in the XSLT 1.0 standard, have no discernable effect at all: indent, media-type, method, omit-xml-declaration, and standalone. All attributes must be set as hardcoded strings.

The cdata-section-elements attribute is a whitespace separated list of qualified node names which should have their text contents enclosed within CDATA sections. This effect is visible in the result tree document copied to the op script, event script, and commit script trace files; however, Junos does not process these CDATA sections, so any text assigned to these elements is ignored during the actual script operation.

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in the result tree document copied to the op script and event script (but not commit script) trace files.

The encoding attribute determines what encoding format should be used for writing the result tree document into the op script and event script (but not commit script) trace files. The following (case-insensitive) values are supported: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii.

The version attribute is used to alter the XML version value that is written in the XML declaration of the result tree document copied into the op script and event script (but not commit script) trace files. It should be set to "1.0", but changing it to other values does not alter the format of the document in any way.

Example

This op script demonstrates the effect seen in the op script trace file when the encoding and doctype-system attributes are set. The copyright symbol © is encoded into the trace file as "©" due to the html encoding.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

<xsl:output encoding="html" doctype-system="local.dtd">;

match / {
    <op-script-results> {
        <output> "Copyright symbol: ©";
    }
}
```

Output

```
jnpr@srx210> op xsl_output
Copyright symbol: ©
```

```
    jnpr@srx210> show log op-script.log
    Apr 9 08:15:18 srx210 clear-log[1884]: logfile cleared
    Apr  9 08:15:22 complete script processing begins
    Apr  9 08:15:22 opening op script '/var/db/scripts/op/xsl_output.slax'
    Apr  9 08:15:22 reading op script 'xsl_output.slax'
    Apr  9 08:15:23 op script processing begins
    Apr  9 08:15:23 reading op script input details
    Apr  9 08:15:23 testing op details
    Apr  9 08:15:23 running op script 'xsl_output.slax'
    Apr  9 08:15:23 opening op script '/var/db/scripts/op/xsl_output.slax'
    Apr  9 08:15:23 reading op script 'xsl_output.slax'
    Apr  9 08:15:23 op script output
    Apr  9 08:15:23 begin dump
    <?xml version="1.0" encoding="html"?>
    <!DOCTYPE op-script-results SYSTEM "local.dtd">
    <op-script-results xmlns:junos="http://xml.juniper.net/junos/*/junos" xmlns:xnm="http://xml.
juniper.net/xnm/1.1/xnm" xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0">
      <output>Copyright symbol: &copy;</output>
    </op-script-results>
    Apr  9 08:15:23 end dump
    Apr  9 08:15:23 inspecting op output 'xsl_output.slax'
    Apr  9 08:15:23 results of op script
    Apr  9 08:15:23 begin dump
    <output>Copyright symbol: &#xA9;</output>Apr  9 08:15:23 end dump
    Apr  9 08:15:23 finished op script 'xsl_output.slax'
    Apr  9 08:15:23 op script processing ends
```

<xsl:processing-instruction>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:processing-instruction name=*"target"***>** *"processing instruction contents"***;**

Description

The **<xsl:processing-instruction>** instruction element is used to add a XML processing instruction to the result tree. This is a rarely used element, as the result tree is processed directly by Junos so there is generally not a reason to use **<xsl:processing-instruction>** within SLAX scripts; however, one possible use case is to add a processing instruction to a XML document that is written to disk through the **<xsl:document>** instruction element, or one of its related extension elements.

The name attribute is mandatory and becomes the target of the processing instruction. Its value is an attribute value template, so it can be set to a hardcoded string, a variable, or a XPath expression. The string contents assigned to the **<xsl:processing-instruction>** element are assigned as the generated processing instruction's contents.

Example

This op script shows how to add a processing instruction to an output XML file.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <xsl:document href="/var/tmp/output.xml" indent="yes" method="xml"> {
            <document-element> {
                <element>;
                <xsl:processing-instruction name="instruction"> "name='testing'";
                <element>;
            }
        }
    }
}
```

Output

```
jnpr@srx210> op xsl_processing-instruction

jnpr@srx210> file show /var/tmp/output.xml
<?xml version="1.0"?>
<document-element>
  <element/>
  <?instruction name='testing'?>
  <element/>
</document-element>
```

<xsl:sort>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:sort data-type="*number|text*" order="*ascending|descending*" select="*expression*"**>;**

Description

The **<xsl:sort>** instruction element sorts the current node list of the **for-each** and **apply-templates** SLAX statements prior to iterating through the node list. To cause sorting to occur, insert the **<xsl:sort>** element immediately after the **for-each** or **apply-templates** statement:

```
for-each( $example/user ) {
    <xsl:sort>;

    loop code...
}


apply-templates user {
    <xsl:sort select="name">;
}
```

Although the location of the **<xsl:sort>** instruction appears to be within the loop code itself, the element is actually only processed when the loop is first initiated. The underlying XML data structure is not permanently sorted, only the order of the current node list being used by the **for-each** or **apply-templates** statement is effected.

If no attributes are provided for **<xsl:sort>** then the list is sorted based on the string value of each node.

Code:

```
var $ns := {
    <user> "lab";
    <user> "admin";
    <user> "read-only";
    <user> "jnpr";
}

<output> "Unsorted:";
for-each( $ns/user ) {
    <output> .;
}

<output> "-------\nSorted:";
for-each( $ns/user ) {
    <xsl:sort>;
    <output> .;
}
```

Output:

```
Unsorted:
lab
admin
read-only
jnpr
-------
Sorted:
admin
jnpr
lab
read-only
```

The select attribute determines each node's comparison string that will be used for sorting. It is set as a hardcoded expression string, so its value is an XPath expression that is evaluated with the node as its context, and then the result is translated into the comparison string for that node. SLAX-specific operators such as == and _ cannot be used within the expression string. If the select attribute is not included then its default value is

“.”, which causes the string content of each node in the list to be compared.

The data-type attribute determines the type of sorting that will be used, and its value is an attribute value template, so it can be set as a hardcoded string, a variable, or a XPath expression, but the only valid options are “text” or “number”. If it is set to “text” then the strings will be compared based on their character values (i.e. ASCII code), so “0” is less than “9”, which is less than “A”, which is less than “Z”, which is less than “a”, which is less than “z”, etc. If data-type is set to “number” then the strings are converted to numbers and compared numerically. The difference between the two types can be seen when comparing “100” with “11”. With ascending text sorting, “100” would come before “11” because “0” has a lower ASCII code than “1”, but with ascending number sorting, “11” would come before “100” because 11 is a smaller number than 100. If data-type is not set then it defaults to “text”.

The order attribute determines if the nodes should be sorted in ascending or descending order. Its value is an attribute value template as well, but it must be either “ascending” or “descending”. If it is not present then the default order is “ascending”.

Multiple **<xsl:sort>** elements can be assigned to a single **for-each** or **apply-templates** statement. They are applied, in order, until a difference is found.

```
apply-templates interfaces/interface/unit {
    <xsl:sort select="../name">;
    <xsl:sort select="name">;
}
```

The XSLT 1.0 specification includes two other attributes for **<xsl:sort>** that are not currently supported in Junos: case-order and lang.

Example

This op script uses **<xsl:sort>** to cause the contents of a node-set to be processed by a **for-each** loop in sorted order.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

match / {
    <op-script-results> {

        /* Example data */
        var $interfaces := {
            <interface> {
                <name> "so-0/0/0";
                <unit> {
                    <name> "0";
                }
            }
            <interface> {
                <name> "ge-1/0/0";
                <unit> {
                    <name> "11";
                }
                <unit> {
                    <name> "6";
                }
```

```
                <unit> {
                    <name> "100";
                }
            }
        }

        /* Sort, first by interface, then by unit */
        for-each( $interfaces/interface/unit ) {
            <xsl:sort select="../name">;
            <xsl:sort select="name" data-type="number">;

            <output> ../name _ "." _ name;
        }
    }
}
```

Output

```
ge-1/0/0.6
ge-1/0/0.11
ge-1/0/0.100
so-0/0/0.0
```

<xsl:text>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:text disable-output-escaping=*"yes|no"***>** *"text content"***;**

Description

The **<xsl:text>** instruction element adds a text node to the result tree. This element is generally not needed within SLAX scripts, as most of its functionality can be achieved through the standard SLAX syntax; however, there is one scenario where **<xsl:text>** would be required: when the script is generating an XSL document through the **<xsl:document>** instruction element, or one of its associated extension elements, and the '<',' >', and '&' characters in the text output should not be escaped.

By default, when outputting a text node to the result tree, or to an output file via the **<xsl:document>** element, in xml or html format all '<', '>', and '&' characters are escaped as "<", ">", or "&", but if the disable-output-escaping attribute is set to "yes", then these characters will not be escaped. The disable-output-escaping attribute is optional, but when included it must be set to a hardcoded string value of either "yes" or "no".

Example

This op script demonstrates how to disable output escaping with the **<xsl:text>** element when writing a XML

document with the **<xsl:document>** element.

Code

```
    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match / {
        <xsl:document href="/var/tmp/output.xml" indent="yes"> {
            <result-tree-document-nodes> {
                <op-script> {
                    <xsl:text disable-output-escaping="yes"> "<![CDATA[<op-script-results>]]>";
                }
                <event-script> {
                    <xsl:text disable-output-escaping="yes"> "<![CDATA[<event-script-results>]]>";
                }
                <commit-script> {
                    <xsl:text disable-output-escaping="yes"> "<![CDATA[<commit-script-
results>]]>";
                }
            }
        }
    }
```

Output

```
    jnpr@srx210> op xsl_text

    jnpr@srx210> file show /var/tmp/output.xml
    <?xml version="1.0"?>
    <result-tree-document-nodes>
      <op-script><![CDATA[<op-script-results>]]></op-script>
      <event-script><![CDATA[<event-script-results>]]></event-script>
      <commit-script><![CDATA[<commit-script-results>]]></commit-script>
    </result-tree-document-nodes>
```

<xsl:value-of>

Source: XSLT
Namespace: http://www.w3.org/1999/XSL/Transform
Common Prefix: xsl
Minimum Version: Junos 8.2

Syntax

<xsl:value-of disable-output-escaping="*yes|no*" select="*expression*"**>;**

Description

The **<xsl:value-of>** instruction element processes the XPath expression included in its select attribute, converts the result to a string, and then adds that string as a text node into the result tree. This element is generally not needed within SLAX scripts, as most of its functionality can be achieved through the SLAX **expr** statement; however, there is one scenario where **<xsl:value-of>** would be required: when the script is generating an XSL document through the **<xsl:document>** instruction element, or one of its associated extension elements, and the '<', '>', and '&' characters in the text output should not be escaped.

By default, when outputting a text node to the result tree, or to an output file via the **<xsl:document>** element, in xml or html format all '<', '>', and '&' characters are escaped as "<", ">", or "&", but if the disable-output-escaping attribute is set to "yes", then these characters will not be escaped. The disable-output-escaping attribute is optional, but when included it must be set to a hardcoded string value of either "yes" or "no".

The select attribute contains the XPath expression to process. It is set as a hardcoded expression string and it cannot include any SLAX-specific syntax such as the == or _ operators.

Example

This op script demonstrates the difference between an escaped and an unescaped XPath result in the output file generated by the **<xsl:document>** element.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Find where the script was enabled */
        var $set-command := { call jcs:grep( $filename = "/var/log/syslog",
                              $pattern = "set system scripts op file xsl_value-of\.slax" ); }

        /* Write to document in escaped and unescaped form */
        <xsl:document href="/var/tmp/output.xml" indent="yes"> {
            <match> {
                <escaped> {
                    if( $set-command//input ) {
                        expr $set-command//input;
                    }
                    else {
                        expr "No match";
                    }
                }
                <unescaped> {
                    if( $set-command//input ) {
                        <xsl:value-of disable-output-escaping="yes"
                            select="$set-command//input">;
                    }
                    else {
                        expr "No match";
```

```
                    }
                }
            }

        }
    }
}
```

Output

```
jnpr@srx210> op xsl_value-of

jnpr@srx210> file show /var/tmp/output.xml
<?xml version="1.0"?>
<match>
  <escaped>&lt;190&gt;1 2011-04-16T07:01:19.879Z srx210 mgd 1213 UI_CMDLINE_READ_LINE
[junos@2636.1.1.1.2.36 username="jnpr" command="set system scripts op file xsl_value-of.slax "]
User 'jnpr', command 'set system scripts op file xsl_value-of.slax '</escaped>
  <unescaped><190>1 2011-04-16T07:01:19.879Z srx210 mgd 1213 UI_CMDLINE_READ_LINE
[junos@2636.1.1.1.2.36 username="jnpr" command="set system scripts op file xsl_value-of.slax "]
User 'jnpr', command 'set system scripts op file xsl_value-of.slax '</unescaped>
</match>
```

<xt:document>

Source: XT
Namespace: http://www.jclark.com/xt
Common Prefix: xt
Minimum Version: Junos 8.2

Syntax

<xt:document append="*yes|no*" doctype-public="*string*" doctype-system="*string*"

encoding="*format*" href="*filename*" indent="*yes|no*" method="*html|text|xml*"

omit-xml-declaration="*yes|no*" standalone="*yes|no*" version="*string*" **>** {

File content...

}

Description

The **<xt:document>** instruction element writes data to an output file on the local disk drive in XML, HTML, or text format. The filename is indicated through the required href attribute, which like all the other attributes of **<xt:document>** is handled as an attribute value template, so its value can be specified as a hardcoded string, a variable, or a XPath expression.

It is recommended that the full path to the output file be provided, because while the default directory for op scripts and commit scripts is /var/tmp, the default directory at times for event scripts is the root directory /, which cannot be written to by **<xt:document>**. The only allowed URI scheme for the filename is "file", and it can be included as part of the href attribute value, but is not required, and there is no difference in behavior if it is not included (e.g. href="file:///var/tmp/output").

The **<xt:document>** element can only write to the local disk. Writing via networking protocols such as FTP or HTTP is not supported, and neither is writing to other routing-engines. Also, the full path specified in the href attribute must already exist because the script processor in Junos does not create new directories as part of the **<xt:document>** operation.

The namespace for **<xt:document>** must be defined as an extension namespace; otherwise, the script processor will not interpret the element as an instruction element:

```
ns xt extension = "http://www.jclark.com/xt";
All contents of the <xt:document> element are written to the output file:
```

Code example:

```
<xt:document href="/var/tmp/example-file" indent="yes"> {
    <chassis> {
        <name> "M40e";
        <name> "EX8216";
    }
}
```

Output file:

```
<?xml version="1.0"?>
<chassis>
  <name>M40e</name>
  <name>EX8216</name>
</chassis>
```

The method attribute can be set to “xml”, “html”, or “text” and it determines the format for the output file. The default method is “xml”; unless the root element of the output file is <html>, in which case the default method is “html”. When method is set to “text”, all of the text content is output, without any escaping or extra indenting.

Code example:

```
var $content = {
    <html> {
        <head> {
            <title> "Example HTML";
        }
        <body> {
            expr "First Line";
            <br>;
            expr "Second Line";
        }
    }
}
<xt:document href="/var/tmp/xml-method" method="xml" indent="yes"> {
    copy-of $content;
}
<xt:document href="/var/tmp/html-method" method="html"> {
    copy-of $content;
}
<xt:document href="/var/tmp/text-method" method="text"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/xml-method
<?xml version="1.0"?>
<html>
  <head>
    <title>Example HTML</title>
  </head>
```

```
  <body>First Line<br/>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/html-method
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Example HTML</title>
</head>
<body>First Line<br>Second Line</body>
</html>

jnpr@srx210> file show /var/tmp/text-method
Example HTMLFirst LineSecond Line
```

By default, the **<xt:document>** element overwrites the output file if it already exists; however, if the append attribute, which was added in Junos 11.1, is set to either "yes" or "true", then the new data will be appended to the file, leaving the existing file contents in place. If append is set to "yes" or "true" when writing using the "xml" method, then the XML declaration is never added, even if the file is being created; however, at the time of this writing, if append is included and is not set to "yes" or "true" then the XML declaration is always added, even if omit-xml-declaration is set to "yes", so it is best to only include the append attribute if it is being enabled, especially since it is off by default.

Code example:

```
<xt:document href="/var/tmp/time-record" method="text"> {
    expr date:date-time() _ "\n";
}
<xt:document href="/var/tmp/time-record" method="text" append="yes"> {
    expr date:date-time() _ "\n";
}
```

Output file:

```
2011-04-23T21:11:31Z
2011-04-23T21:11:31Z
```

A large caveat exists when using **<xt:document>**; it always accesses the file system as user "nobody". This means that files cannot be overwritten or appended unless their permissions give write access to user "nobody" or to everyone, and files cannot be created unless the directory provides similar permissions. In addition, when the **<xt:document>** element creates an output file, it is owned by user "nobody" and its file permissions are 644, meaning that while anyone can read it, only user "nobody" can edit or delete it. (As of Junos 10.0R3, super-users can delete these files from the CLI). As a workaround, create the file initially through the <file-put> RPC with permissions 666, giving everyone read and write access, and then have **<xt:document>** overwrite the original file. As a result, the file will be owned by the executing script's user and will have the desired contents as created by **<xt:document>**.

Example code:

```
var $rpc = {
    <get-configuration> {
        <configuration> {
            <event-options> {
                <destinations>;
            }
        }
    }
}
<xt:document href="/var/tmp/destination-config" indent="yes"> {
    copy-of jcs:invoke( $rpc );
}
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-r--r--  1 nobody wheel       405 Apr 23 22:01 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

Example code:

```
    var $rpc = {
        <get-configuration> {
            <configuration> {
                <event-options> {
                    <destinations>;
                }
            }
        }
    }
    var $filename = "/var/tmp/destination-config";
    var $put-rpc = {
        <file-put> {
            <filename> $filename;
            <permission> "666";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "To be overwritten";
        }
    }
    var $results = jcs:invoke( $put-rpc );
    <xt:document href=$filename indent="yes"> {
        copy-of jcs:invoke( $rpc );
    }
```

Output:

```
    jnpr@srx210> file list /var/tmp/destination-config detail
    -rw-rw-rw-  1 jnpr  staff        405 Apr 23 22:07 /var/tmp/destination-config
    total 1

    jnpr@srx210> file show /var/tmp/destination-config
    <?xml version="1.0"?>
    <configuration xmlns:junos="http://xml.juniper.net/junos/*/junos" junos:changed-
seconds="1303539992" junos:changed-localtime="2011-04-23 06:26:32 UTC">
        <event-options>
            <destinations>
                <name>local</name>
                <archive-sites>
                    <name>/var/tmp</name>
                </archive-sites>
            </destinations>
        </event-options>
    </configuration>
```

The doctype-public and doctype-system attributes can be used to cause a DOCTYPE declaration to appear in

the output file:

Code example:

```
<xt:document href="/var/tmp/output" doctype-system="local.dtd"> {
    <parent> {
        <child>;
        <child>;
        <child>;
    }
}
```

Output file:

```
<?xml version="1.0"?>
<!DOCTYPE parent SYSTEM "local.dtd">
<parent><child/><child/><child/></parent>
```

The encoding attribute determines what character encoding format should be used when writing the output file. The following (case-insensitive) values can be used: ascii, html, iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-13, iso-8859-14, iso-8859-15, iso-8859-16, iso-latin-1, iso-latin-2, utf8, utf-8, utf16, utf-16, utf-16le, utf-16be, and us-ascii. The default value is utf-8.

Code example:

```
<xt:document href="/var/tmp/output-html" encoding="html"> {
    <copyright> "©";
}
```

Output file:

```
<?xml version="1.0" encoding="html"?>
<copyright>&copy;</copyright>
```

The indent attribute can be set to either “yes” or “no”, and it determines whether indentation should be performed or not. It is off by default for the “xml” method and on by default for the “html” method. It has no effect on text files.

Code example:

```
var $content = {
    <interfaces> {
        <interface> {
            <name> "ge-0/0/0";
        }
    }
}
<xt:document href="/var/tmp/output1" method="xml"> {
    copy-of $content;
}
<xt:document href="/var/tmp/output2" method="xml" indent="yes"> {
    copy-of $content;
}
```

Output files:

```
jnpr@srx210> file show /var/tmp/output1
<?xml version="1.0"?>
<interfaces><interface><name>ge-0/0/0</name></interface></interfaces>

jnpr@srx210> file show /var/tmp/output2
<?xml version="1.0"?>
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
```

```
  </interface>
</interfaces>
```

A XML declaration is included by default when using the "xml" method (unless appending is enabled). To disable the XML declaration, set the omit-xml-declaration attribute to "yes". This attribute has no effect when using the "html" or "text" methods.

Code example:

```
<xt:document href="/var/tmp/output" indent="yes"
               omit-xml-declaration="yes"> {
    <details> {
        <script> $script;
        <user> $user;
    }
}
```

Output file:

```
<details>
  <script>test.slax</script>
  <user>jnpr</user>
</details>
```

By default, no standalone document declaration is included in the XML declaration, but if the standalone attribute is set to either "yes" or "no", then standalone is included in the declaration with the specified value.

Code example:

```
<xt:document href="/var/tmp/context" indent="yes" standalone="yes" > {
    copy-of $junos-context/user-context;
}
```

Output file:

```
<?xml version="1.0" standalone="yes"?>
<user-context xmlns:junos="http://xml.juniper.net/junos/*/junos">
  <user>jnpr</user>
  <class-name>j-super-user</class-name>
  <uid>2001</uid>
  <login-name>jnpr</login-name>
</user-context>
```

The version attribute is used to alter the XML version value that is written in the XML declaration of the output file when the "xml" method is used. It should be set to "1.0", but changing it to other values does not alter the format of the document in any way.

The **<xt:document>** element has an additional attribute: cdata-section-elements, which is intended to indicate what nodes in the XML output file should have their text contents enclosed within CDATA sections, but it is non-functional at the time of this writing. Also, the media-type attribute is not implemented and has no effect.

If an error occurs due to an inability to write to the output file, an incorrect attribute setting, or for some other reason, then the script fails and an error message is displayed to the script user for op scripts and commit scripts or to the event script output file. The script itself is unable to react to the error.

The Junos API contains a RPC element for writing to the disk: <file-put>, which has some capabilities that **<xt:document>** does not have, including the ability to write in base64 format, to select file permissions, and to write to the non-local routing-engines. In addition, unlike **<xt:document>**, the <file-put> RPC accesses the file system using the credentials of the user that is executing the script. However, <file-put> is incapable of writing in XML or HTML format or of appending to an existing file, so when a script needs to perform one of those three operations then **<xt:document>** would be an appropriate element to use.

Junos contains support for five instruction elements that have equivalent functionality: **<exsl:document>**,

<redirect:write>, <saxon:output>, <xsl:document>, and **<xt:document>. All attributes and functionality are common among the five elements, except for some variances in support for the different file selector attributes: file, href, and select. Given that all five elements perform the same operation, it is recommended that the <xsl:document> element be used in place of the others because, unlike the other four, it does not need to have an extension namespace defined.**

Example

This op script demonstrates how to use the five file writing instruction elements.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns exsl extension = "http://exslt.org/common";
ns saxon extension = "http://icl.com/saxon";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";
ns xt extension = "http://www.jclark.com/xt";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* <redirect:write> <xsl:document> <saxon:output> and appending */
        var $append-file = "/var/tmp/append-example";
        <redirect:write href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <xsl:document href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }
        <saxon:output href=$append-file append="yes"> {
            copy-of $junos-context/localtime-iso;
        }

        /* <exsl:document> and indenting */
        <exsl:document href="/var/tmp/indent-example" indent="yes"> {
            <chassis> {
                <platform> "m40e";
                <platform> "mx960";
                <platform> "ex8216";
            }
        }

        /* <xt:document> and text output */
        <xt:document href="/var/tmp/text-example" method="text"> {
            expr "Class-name: " _ $junos-context/user-context/class-name _ "\n";
            expr "User-name: " _ $junos-context/user-context/user _ "\n";
        }
    }
}
```

Output

```
jnpr@srx210> file list detail /var/tmp/*-example
-rw-r--r--  1 nobody wheel       318 Apr 25 15:31 /var/tmp/append-example
```

```
    -rw-r--r--  1 nobody wheel       130 Apr 25 15:31 /var/tmp/indent-example
    -rw-r--r--  1 nobody wheel        41 Apr 25 15:31 /var/tmp/text-example
    total 3

    jnpr@srx210> file show /var/tmp/append-example
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>
    <localtime-iso xmlns:junos="http://xml.juniper.net/junos/*/junos">2011-04-25 15:31:44 UTC</
localtime-iso>

    jnpr@srx210> file show /var/tmp/indent-example
    <?xml version="1.0"?>
    <chassis>
      <platform>m40e</platform>
      <platform>mx960</platform>
      <platform>ex8216</platform>
    </chassis>

    jnpr@srx210> file show /var/tmp/text-example
    Class-name: j-super-user
    User-name: jnpr
```

Part 5: Templates

This section documents the templates that are available in the junos.xsl import file, which is imported into all scripts that follow the standard boilerplate. Internal templates that only serve to fulfill the task of the public templates are not documented. Internal parameters, which are not intended to be set by the script writer, are also not documented.

The displayed syntax mentions no return value because templates always write any output to the result tree, rather than returning a data type. This can, however, be redirected into a variable if desired. See the description of the **call** statement for details.

jcs:edit-path

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

jcs:edit-path($dot = .)

Description

The **jcs:edit-path** template adds an <edit-path> element to the result tree. <edit-path> elements provide configuration hierarchy context to <xnm:warning> and <xnm:error> messages through their text content, which is in the following format:

```
<edit-path> "[edit system syslog]"
```

This is an example of a standard <xnm:warning> message, without an <edit-path> assigned:

```
<xnm:warning> {
    <message> "Telnet is enabled";
}
```

It displays the following upon commit:

```
[edit]
jnpr@srx210# commit
warning: Telnet is enabled
commit complete
```

But an <edit-path> element can be added to provide more information:

```
<xnm:warning> {
    <edit-path> "[edit system services]";
    <message> "Telnet is enabled";
}
```

And now, the problem hierarchy is displayed:

```
[edit]
jnpr@srx210# commit
[edit system services]
  warning: Telnet is enabled
commit complete
```

While it is possible to add <edit-path> elements manually, the **jcs:edit-path** template provides a simpler method as it builds them automatically based on either the current context position, or an alternate position specified by the $dot parameter, which can be assigned a node-set value:

```
<xnm:warning> {
    call jcs:edit-path( $dot = system/services );
    <message> "Telnet is enabled";
}
```

However, the created <edit-path> string might not match the path string seen within the CLI in all cases. For example, the path string shown while configuring OSPF virtual-links looks like the following:

```
[edit protocols ospf area 0.0.0.0 virtual-link neighbor-id 10.0.0.1 transit-area 0.0.0.0]
jnpr@srx210#
```

But the **jcs:edit-path** template is unaware of identifiers that are not named <name>, so it creates the following <edit-path> element:

```
<edit-path> "[edit protocols ospf area 0.0.0.0 virtual-link]"
```

Another difference can be seen with configuration elements that are typically hidden in the CLI path string, such as the <interface> element, which looks like the following in the CLI:

```
[edit interfaces ge-0/0/0]
jnpr@srx210#
```

But the created <edit-path> element looks like the following, because **jcs:edit-path** has no way to know that <interface> should be not be displayed:

```
<edit-path> "[edit interfaces interface ge-0/0/0]"
```

If jcs:edit-path is called for a non-existent configuration hierarchy then the following <edit-path> will be created:

```
<edit-path> "[unknown location]"
```

Note that jcs:edit-path is only intended for nodes within a standard Junos XML configuration hierarchy, and as of the time of this writing, attempting to use it with a different XML hierarchy causes the commit script to fail with a recursion error.

Example

This commit script example demonstrates how to use **jcs:edit-path** both with and without setting the $dot parameter.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {

    /* Provide warnings for any user accounts without authentication configured */
    for-each( system/login/user ) {
        if( jcs:empty( authentication ) ) {
            <xnm:warning> {
                /*  $dot does not need to be set because the context node is already
                    at the desired location. */
                call jcs:edit-path();
                <message> "Authentication is not configured.";
            }
        }
    }

    /* Verify that SSH is enabled */
    if( jcs:empty( system/services/ssh ) ) {
        <xnm:warning> {
            /* $dot must be set to the correct context */
            call jcs:edit-path( $dot = system/services );
            <message> "SSH is not enabled.";
        }
    }
}
```

Output

```
[edit]
jnpr@srx210# commit check
[edit system login user new-user]
  warning: Authentication is not configured.
[edit system services]
  warning: SSH is not enabled.
configuration check succeeds
```

jcs:emit-change

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

jcs:emit-change($content, $dot = ., $message, $tag = “change”)

Description

The **jcs:emit-change** template adds either a <change> or <transient-change> element to the result tree.

These two result tree elements are used by commit scripts to perform permanent or transient configuration changes, respectively. They can be built manually, but the advantage of the **jcs:emit-change** template is that it automatically creates the appropriate configuration hierarchy for the change elements, allowing script writers to indicate configuration changes that are relative to a certain configuration location instead of specifying the complete hierarchy. (This template has no effect in op scripts and event scripts, because they cannot use the <change> or <transient-change> result tree elements. Instead, they can make changes through the **jcs:load-configuration** template).

The configuration change instructions, which are expressed in XML, must be assigned to the $content parameter. The change instructions need to be relative to the $dot parameter, which defaults to the current configuration context if not provided, but can be set to an alternate configuration node if appropriate.

For example, the following commit script code generates the instructions to delete the [edit system syslog] hierarchy:

```
var $change = {
    <syslog delete="delete">;
}
call jcs:emit-change( $content = $change, $dot = system );
```

This is the <change> element that is written to the result tree by **jcs:emit-change** as a result of the above code:

```
<change>
    <system>
        <syslog delete="delete"/>
    </system>
</change>
```

If the $message parameter is set then a <xnm:warning> element is added to the result tree with the $message

string as its message. As part of this warning element creation, **jcs:emit-change** calls the **jcs:edit-path** template to generate an <edit-path> element, which is always included as part of the warning message.

Note: As of the time of this writing, the jcs:edit-path template is called without setting its $dot parameter, which means that the <edit-path> will always be generated based on the current configuration context rather than the $dot value supplied to jcs:emit-change. This could result in an incorrect <edit-path> or even a script recursion error. Because of this, it is best to not include the $message parameter anytime the $dot parameter is used; instead, the desired <xnm:warning> element should be created manually.

Here is a modification of the initial configuration example, which deletes the [edit system syslog] hierarchy, but this time a message is included:

```
    var $change = {
        <syslog delete="delete">;
    }
    call jcs:emit-change( $content = $change, $message = "Removing syslog config", $dot = system
);
```

With the above code, the following result tree elements are generated by **jcs:emit-change**:

```
    <xnm:warning>
        <edit-path>[edit]</edit-path>
        <message>Removing syslog config</message>
    </xnm:warning>
    <change>
        <system>
            <syslog delete="delete"/>
        </system>
    </change>
```

Note: as mentioned previously, the **<edit-path>** in the output above does not have the correct hierarchy because jcs:emit-change currently does not provide the $dot parameter to jcs:edit-path.

The $tag parameter indicates the name of the result tree element that should be created. It defaults to "change", causing a **<change>** element to be made, but it can be set to "transient-change" instead, in order to create a <transient-change> element.

Note: jcs:emit-change is unable to work within configuration hierarchies that use identifiers other than **<name>** (for example: OSPF virtual-links, which have two identifiers, neither of which are **<name>**), and the created **<change>** or **<transient-change>** element for these hierarchies will cause a commit error. In these scenarios, it is necessary to create the <change> manually instead of relying on jcs:emit-change.

Example

This commit script performs two configuration changes by using the **jcs:emit-change** template.

Code

```
    version 1.0;

    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match configuration {

        /* If SSH is not enabled, then add it
           and if Telnet is enabled then remove it */
```

```
    if( jcs:empty( system/services/ssh ) ) {
        var $change = {
            <ssh>;
        }
        call jcs:emit-change( $content = $change, $dot = system/services );
        <xnm:warning> {
            <message> "Enabling SSH";
        }
    }
    if( system/services/telnet ) {
        var $change = {
            <telnet delete="delete">;
        }
        call jcs:emit-change( $content = $change, $dot = system/services );
        <xnm:warning> {
            <message> "Disabling Telnet";
        }
    }
}
```

Output

```
[edit]
jnpr@srx210# commit check
warning: Enabling SSH
warning: Disabling Telnet
configuration check succeeds
```

jcs:emit-comment

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

jcs:emit-comment()

Description

The **jcs:emit-comment** template generates a <junos:comment> element. The comment includes the user that performed the action and the time, as well as the script, as shown in this example:

```
<junos:comment> "/* Added by user jnpr on Wed Mar  2 15:46:22 2011 (script test.slax) */"
```

To take effect, the <junos:comment> element must appear within the configuration change instructions, and it must immediately precede the configuration element it should apply to. Here is an example of a commit script <change> element that has embedded the **jcs:emit-comment** template within it in order to annotate who added the new user account and when it was added:

```
<change> {
    <system> {
        <login> {
            call jcs:emit-comment();
            <user> {
```

```
                <name> "new-user";
                <class> "read-only";
                <authentication> {
                    <encrypted-password> "$1$.qqAHsnM$PF3MB3KAJ2CUti7cOPILX1";
                }
            }
        }
    }
```

As a result of this change, the following comment is seen on the user account:

```
/* Added by user jnpr on Wed Mar  2 15:41:26 2011 (script test.slax) */
user new-user {
    uid 2003;
    class read-only;
    authentication {
        encrypted-password "$1$.qqAHsnM$PF3MB3KAJ2CUti7cOPILX1";
    }
}
```

Example

This op script uses **jcs:emit-comment** to annotate the details behind the new static route configurations that it applies.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        /* Gather information */
        var $route = jcs:get-input( "Enter route destination: " );
        var $nh = jcs:get-input( "Enter next-hop: " );

        var $configuration-change = {
            <configuration> {
                <routing-options> {
                    <static> {
                        call jcs:emit-comment();
                        <route> {
                            <name> $route;
                            <next-hop> $nh;
                        }
                    }
                }
            }
        }
        var $connection = jcs:open();
        var $results := { call jcs:load-configuration( $connection, $configuration =
$configuration-change ); }
        /* Display any errors/warnings */
        copy-of $results;
        expr jcs:close( $connection );
    }
```

```
}
```

Output

```
Enter route destination: 192.168.0.0/16
Enter next-hop: 10.0.0.1

jnpr@srx210> show configuration routing-options
static {
    /* Added by user jnpr on Wed Mar  2 16:00:11 2011 (script jcs_emit-comment.slax) */
    route 192.168.0.0/16 next-hop 10.0.0.1;
}
```

jcs:grep

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.0

Syntax

jcs:grep($filename, $pattern)

Description

The **jcs:grep** template provides an easy way to search an ascii file for lines matching a regular expression pattern. The file is set through the $filename parameter, and the regular expression is assigned through the $pattern parameter, both of which are strings. The template works by reading the file through "file show", breaking it into its component lines, and then providing each line to the **jcs:regex**() function, looking for matches to the provided pattern.

It is best to provide a full path for the $filename parameter, but in Junos 11.1 and beyond, the following default directories are applied to a relative file name. (Prior to 11.1 there was no default directory):

Script Type	Default Directory
Op script	Script user's home directory
Commit script	/var/tmp
Event script (Enabled under event-options)	/var/tmp
Event script (Enabled under system)	/

The supported regular expression syntax is discussed in the description of the **jcs:regex**() function. The pattern matching is performed on a per-line basis, so it is not possible to match a pattern spread over multiple lines.

Note: A blank line is automatically added to the beginning of the text file by "file show", so a regular expres-

sion matching on blank lines will return an extra match.

If there is an error with the regular expression syntax then an error is displayed to the screen for every line, and no result is returned; however, if the file does not exist or the script user lacks the permissions to access it, then no result is returned and no error is displayed.

Matching lines are written to the result tree in the following format:

```
<match> {
    <input> Matching line from file
    <output> Matching string from line
}
```

Function call:

```
    jcs:grep( $filename = "/var/log/messages" , $pattern = "Starting of initial processes
complete")
    Result tree fragment:
    <match> {
        <input> "Mar  4 15:23:43  srx210 init: Starting of initial processes complete";
        <output> "Starting of initial processes complete";
    }
```

Because it is a template, **jcs:grep** writes its output to the result tree, which should be redirected into a node-set variable so that it can be used by the script. This can be done by using the := node-set conversion operator, as shown in this example:

```
var $filename = "/var/log/messages";
var $pattern =
    "^[[:alpha:]]{3}[ ]{1,2}[0-9]{1,2} [0-9]{2}:[0-9]{2}:[0-9]{2}";
var $result := { call jcs:grep( $filename, $pattern ); }
```

Because $result is a node-set variable, pointing at the root node of the converted result tree fragment from **jcs:grep**, location paths can be used to extract the assembled information:

```
for-each( $result//output ) {
    <output> "Matched: " _ .;
}
```

Example

This op script shows how to use **jcs:grep** to extract the desired lines from a log file.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-result> {
        var $filename = "/var/log/messages";
        var $pattern = "UI_LOGIN_EVENT";
        var $result := { call jcs:grep( $filename, $pattern ); }

        <output> "Login lines from " _ $filename;
        for-each( $result//input ) {
            <output> .;
```

```
            }
        }
    }
```

Output

```
    Login lines from /var/log/messages
    Mar  5 08:17:00  srx210 file[1277]: UI_LOGIN_EVENT: User 'jnpr' login, class 'j-super-user'
[1277], ssh-connection '10.0.0.50 61274 10.0.0.10 22', client-mode 'junoscript'
    Mar  5 08:33:35  srx210 mgd[1289]: UI_LOGIN_EVENT: User 'jnpr' login, class 'j-super-user'
[1289], ssh-connection '10.0.0.50 61382 10.0.0.10 22', client-mode 'cli'
    Mar  5 08:33:46  srx210 mgd[1294]: UI_LOGIN_EVENT: User 'test' login, class 'j-read-only'
[1294], ssh-connection '10.0.0.50 61383 10.0.0.10 22', client-mode 'cli'
    Mar  5 08:36:46  srx210 file[1299]: UI_LOGIN_EVENT: User 'jnpr' login, class 'j-super-user'
[1299], ssh-connection '10.0.0.50 61274 10.0.0.10 22', client-mode 'junoscript'
```

jcs:load-configuration

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 9.3

Syntax

jcs:load-configuration($connection, $configuration, $action = “merge”, $commit-options)

Description

The **jcs:load-configuration** template simplifies configuration changes for op scripts and event scripts by performing all the necessary steps: locking the configuration database, loading the configuration change, committing the configuration, and unlocking the configuration database. In addition, appropriate error checking halts the commit process if an error occurs while locking the database or loading the configuration. (This template does not work within commit scripts, which make changes through the <change> and <transient-change> result tree elements instead.)

The $connection parameter must be assigned to a connection handle that has been returned by the **jcs:open**() function. It can refer to either a local connection, when making configuration changes to the local Junos device, or it could be a connection to a remote Junos device.

The $configuration parameter should be set to a XML representation of the configuration change that is desired and must include a base element of <configuration>, but either a node-set or result tree fragment data type is acceptable.

This shows an example of a configuration change that sets the hostname to “Junos”:

```
var $connection = jcs:open();
var $configuration = {
    <configuration> {
        <system> {
            <host-name> "Junos";
        }
    }
```

```
}
var $results := {
    call jcs:load-configuration( $connection, $configuration ); }
```

The $action parameter can be set to either “merge”, “replace”, “override”, or “update”, and indicates which load behavior to use for loading the configuration: load merge, load replace, load override, or load update. The default behavior when the $action parameter is not specified is merge.

- merge – Adds the new configuration elements and hierarchies into the existing configuration with new statements overriding old ones.
- replace – Operates the same as a load merge except that any hierarchies tagged with an attribute of replace=“replace” in the new configuration will completely replace the matching hierarchies in the existing configuration.
- override – Completely replaces the existing configuration with the new configuration.
- update – Completely replaces the existing configuration, similar to override, but does not change existing configuration elements and hierarchies that match the new configuration so that Junos daemons are only notified of actual changes between the existing and new configuration.

This example sets the $action parameter to “replace” and includes a replace=“replace” attribute on the <routing-options> element node to cause the entire existing routing-options hierarchy to be replaced by the routing-options hierarchy in the new configuration:

```
var $connection = jcs:open();
var $configuration = {
    <configuration> {
        <routing-options replace="replace"> {
            <static> {
                <route> {
                    <name> "0.0.0.0/0";
                    <next-hop> "10.0.0.1";
                }
            }
        }
    }
}
var $results := { call jcs:load-configuration( $connection, $configuration,
                                               $action = "replace" ); }
```

The **jcs:load-configuration** template makes its changes in configuration exclusive mode, and it only requests a standard “commit” by default. If a different commit option is desired, such as “commit synchronize”, then the $commit-options parameter can be specified. It was introduced in Junos 10.2, can be either a node-set or a result tree fragment data type, and can include any of the following elements as child elements of <commit-options>:

- <check> - Perform a “commit check” only.
  - At the time of this writing, no success message is provided in response to a check, but errors are reported, so the lack of errors should be used to indicate success.
- <full> - Perform a “commit full”.
- <log> “commit comment” – Include a comment for the commit.
- <synchronize> - Perform a “commit synchronize”.
- <force-synchronize> - Equivalent to “commit synchronize force”, enforces the changes on the other routing engine even in the presence of uncommitted configuration changes.

For example, the following code would perform a “commit synchronize” and include a comment while adding

a new user to the configuration:

```
    var $connection = jcs:open();
    var $configuration = {
        <configuration> {
            <system> {
                <login> {
                    <user> {
                        <name> "new-user";
                        <class> "super-user";
                        <authentication> {
                            <encrypted-password> "$1$6/48C1MP$O4sOhtE.rIalMH.7iYm.T1";
                        }
                    }
                }
            }
        }
    }
    var $commit-options = {
        <commit-options> {
            <synchronize>;
            <log> "Adding new-user";
        }
    }
    var $result := { call jcs:load-configuration( $connection, $configuration, $commit-options );
}
```

As it is a template, **jcs:load-configuration** writes to the result tree, which can be redirected to a variable in order for the script to check for warnings or errors:

```
    var $result := { call load-configuration( $connection, $configuration); }
```

In the above example, the := node-set conversion operator is used to ensure that the $result variable is a node-set rather than a result tree fragment. This makes it possible to use location paths to scan for warnings or errors.

Any warnings or errors that occur when locking the database, loading the configuration, committing the configuration, or unlocking the database will be written to the result tree as <xnm:warning> or <xnm:error> elements.

At the time of this writing, the commit RPC is incorrectly copied to the result tree and should be ignored. Also, there is no message provided indicating that the commit was successful, so the absence of <xnm:error> elements should be interpreted as success.

Example

This op script uses **jcs:load-configuration** to add a new user to the configuration. A comment is included with the commit to log that the user was added.

Code

```
    version 1.0;

    ns junos = "http://xml.juniper.net/junos/*/junos";
    ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
    ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

    import "../import/junos.xsl";

    match / {
        <op-script-results> {
```

```
        /* Retrieve values */
        var $user-name = jcs:get-input( "Enter user name: " );
        var $password = jcs:get-secret( "Enter password: " );

        var $connection = jcs:open();
        var $configuration = {
            <configuration> {
                <system> {
                    <login> {
                        <user> {
                            <name> $user-name;
                            <class> "super-user";
                            <authentication> {
                                <plain-text-password-value> $password;
                            }
                        }
                    }
                }
            }
        }
        var $commit-options = {
            <commit-options> {
                <log> "Adding new user: " _ $user-name;
            }
        }
        var $result := {
            call jcs:load-configuration( $connection, $configuration, $commit-options ); }

        /* Report errors or success */
        if( $result//self::xnm:error ) {
            copy-of $result;
        }
        else {
            /* Report any warnings */
            copy-of $result//self::xnm:warning;
            <output> "commit complete";
        }
    }
}
```

Output

```
Enter user name: new-admin
Enter password:
commit complete

jnpr@srx210> show system commit
0   2011-03-07 16:17:57 UTC by jnpr via junoscript
    Adding new user: new-admin
```

jcs:statement

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

jcs:statement($dot = .)

Description

The **jcs:statement** template adds a <statement> element to the result tree. These elements provide additional context to <xnm:warning> and <xnm:error> messages through their text content, which is in the following format:

```
<statement> "mtu 1500;"
```

This is an example of a standard <xnm:warning> message with an <edit-path> created by the **jcs:edit-path** template but without a <statement> assigned:

```
<xnm:warning> {
    call jcs:edit-path();
    <message> "Static route is pointing at old gateway";
}
```

It displays the following upon commit:

```
[edit]
jnpr@srx210# commit
[edit routing-options static route 0.0.0.0/0]
  warning: Static route is pointing at old gateway
commit complete
```

But a <statement> element could make the warning more clear, so the following example also uses the **jcs:statement** template to automatically create a <statement> for the faulty next-hop:

```
<xnm:warning> {
    call jcs:edit-path();
    call jcs:statement( $dot = next-hop );
    <message> "Static route is pointing at old gateway";
}
```

And now, the problem next-hop statement is also displayed:

```
[edit]
jnpr@srx210# commit
[edit routing-options static route 0.0.0.0/0]
    'next-hop 10.0.0.1;'
    warning: Static route is pointing at old gateway
commit complete
```

It is possible to create <statement> elements manually, but using the **jcs:statement** template is a simpler method as it creates them automatically based on either the current context node, or on an alternate context node specified by the $dot parameter, which can be assigned a node-set value. In the above example, the context node was <route>, but the <statement> needed to refer to the <next-hop> instead, so $dot was set to <next-hop>, causing the correct <statement> element to be generated.

Example

This commit script uses **jcs:statement** to indicate the problem policy in its <xnm:warning> message.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
```

```
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {

    var $approved-set := {
        <policy> "remove-local-pref";
        <policy> "apply-communities";
        <policy> "block-local";
        <policy> "block-private";
    }

    /* Scan through BGP import policies and warn if any appear that are unapproved.
       In this scenario, only scan for policies defined under the BGP neighbor, although
       it is possible that policies could have been defined at the group or bgp level. */
    for-each( protocols/bgp/group/neighbor/import ) {
        if( jcs:empty( $approved-set/policy[ . == current() ] ) ) {
            <xnm:warning> {
                call jcs:edit-path( $dot = .. );
                call jcs:statement();
                <message> "Unapproved BGP import policy in use";
            }
        }
    }
}
```

BGP configuration

```
protocols {
    bgp {
        group EBGP {
            peer-as 65501;
            neighbor 10.0.0.2 {
                import [ block-private adjust-med ];
            }
        }
    }
}
```

Output

```
[edit]
jnpr@srx210# commit
[edit protocols bgp group EBGP neighbor 10.0.0.2]
  'import adjust-med;'
    warning: Unapproved BGP import policy in use
commit complete
```

Part 6: Default Parameters

In addition to the templates mentioned in the prior section, the junos.xsl import file also defines six global parameters and one global variable that are available to all script types that follow the standard boilerplate. These default global parameters and variable are covered in this section.

Unlike the other sections, there is no syntax included in the descriptions because the name of the parameter or variable itself is the syntax, and its use is demonstrated in the example that accompanies each description.

$hostname

Minimum Version: Junos 8.2

Description

The **$hostname** global parameter is a string containing the hostname of the local routing-engine that the script is running on. Note the difference in the presence of a hyphen between the **$hostname** parameter and the host-name configuration element:

```
system {
    host-name example;
}
```

Example

This op script example displays the current hostname.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$hostname = " _ $hostname;
    }
}
```

Output

```
$hostname = srx210
```

$localtime

Minimum Version: Junos 8.2

Description

The **$localtime** global parameter is a string containing the local date and time when the script was executed. It is provided in the following format: DDD MMM dd hh:mm:ss YYYY

- DDD – Day name abbreviation
  - Sun, Mon, Tue, Wed, Thu, Fri, Sat
- MMM – Month abbreviation

- Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
- dd – Day of the month
  - Less than ten is preceded by a space rather than a zero.
- hh – Hour of the day
  - Less than ten is preceded by a zero.
- mm – minute of the hour
  - Less than ten is preceded by a zero.
- ss – second of the minute
  - Less than ten is preceded by a zero.
- YYYY – Year

The system's time-zone is not indicated in the **$localtime** string.

Example values:

```
Thu May  5 12:40:27 2011
Sat Jan  1 00:00:05 2000
Sat Dec 25 01:01:04 2010
Sun Dec 25 01:01:26 1960
```

Example

This op script example displays the local date and time when the script is being executed.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$localtime = " _ $localtime;
    }
}
```

Output

```
$localtime = Thu May  5 01:10:13 2011
```

$localtime-iso

aka $localtime_iso
Minimum Version: Junos 9.6

($localtime_iso: 8.4)

Description

The **$localtime-iso** global parameter is a string containing the local date and time when the script was executed. Despite its name, the **$localtime-iso** parameter's format has a couple differences from the ISO 8601 time format: there is no "T" separating the date and time, and the time-zone is provided as an abbreviation instead of an offset.

This is the format followed by the **$localtime-iso parameter**: YYYY-MM-DD hh:mm:ss TZN

- YYYY – Year
  - Always four characters
- MM – Month
  - Always two characters
- DD – Day
  - Always two characters
- hh – Hour
  - Always two characters
- mm – Minute
  - Always two characters
- ss – Second
  - Always two characters
- TZN – Time-zone abbreviation
  - UTC, etc.

For a date and time string that more closely follows the ISO 8601 standard, use the **date:date-time**() function.

The **$localtime-iso** parameter was originally added in Junos 8.4 as **$localtime_iso**, but was changed to **$localtime-iso** in Junos 9.6; however, the old parameter name continues to be provided to scripts for backward compatibility.

Example values:

```
1999-01-02 03:04:21 MST
2010-12-25 10:00:06 UTC
2011-05-05 12:35:07 UTC
```

Example

This op script example displays the local date and time when the script is being executed.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$localtime-iso = " _ $localtime-iso;
    }
}
```

Output

```
$localtime-iso = 2011-05-05 12:36:25 UTC
```

$junos-context

Minimum Version: Junos 11.1

Description

Unlike the rest of the topics discussed in this section, **$junos-context** is actually a global variable rather than a global parameter. This distinction, however, is not important because the **$junos-context** variable can be used throughout a script in the same way as all of the default parameters.

$junos-context is a node-set that contains a single node: the <junos-context> node within the script's source tree. This is a new hierarchy added to the source tree of all script types beginning in Junos 11.1 (and Junos 10.4R2). It is accessible through a location path based on the script type: /commit-script-input/junos-context, /event-script-input/junos-context, or /op-script-input/junos-context. The **$junos-context** variable is defined within the junos.xsl import file as a union of these three location paths, so it always refers to the <junos-context> node whether the script is a commit script, an event script, or an op script. Using this variable, rather than a location path, to access the <junos-context> information is more reliable because it can be used even in portions of the script where the current node is located in a different XML document than the source tree

The <junos-context> hierarchy contains a number of common element nodes that are present in all three script types, along with the <user-context> element node that is available in all script types as well. In addition, commit scripts and op scripts have unique hierarchies that are present within <junos-context>: <commit-context> and <op-context>.

Op script example:

```
<junos-context>
<hostname>srx210</hostname>
<product>srx210h</product>
<localtime>Wed Aug 18 09:04:30 2010</localtime>
<localtime-iso>2010-08-18 09:04:30 UTC</localtime-iso>
<script-type>op</script-type>
<pid>1643</pid>
<tty>/dev/ttyp0</tty>
<chassis>others</chassis>
<routing-engine-name>re0</routing-engine-name>
<re-master/>
<user-context>
    <user>jnpr</user>
    <class-name>j-super-user</class-name>
    <uid>2001</uid>
```

```
        <login-name>jnpr</login-name>
    </user-context>
    <op-context>
        <via-url/>
    </op-context>
</junos-context>
```

Event script example:

```
    <junos-context>
    <hostname>srx210</hostname>
    <product>srx210h</product>
    <localtime>Wed Aug 18 14:25:00 2010</localtime>
    <localtime-iso>2010-08-18 14:25:00 UTC</localtime-iso>
    <script-type>event</script-type>
    <pid>1587</pid>
    <chassis>others</chassis>
    <routing-engine-name>re0</routing-engine-name>
    <re-master/>
    <user-context>
        <user>root</user>
        <class-name>super-user</class-name>
        <uid>0</uid>
        <login-name>root</login-name>
    </user-context>
</junos-context>
```

Commit script example:

```
    <junos-context>
    <hostname>srx210</hostname>
    <product>srx210h</product>
    <localtime>Wed Aug 18 12:30:20 2010</localtime>
    <localtime-iso>2010-08-18 12:30:20 UTC</localtime-iso>
    <script-type>commit</script-type>
    <pid>3003</pid>
    <tty>/dev/ttyp1</tty>
    <chassis>others</chassis>
    <routing-engine-name>re0</routing-engine-name>
    <re-master/>
    <user-context>
        <user>jnpr</user>
        <class-name>j-super-user-local</class-name>
        <uid>2001</uid>
        <login-name>jnpr</login-name>
    </user-context>
    <commit-context>
        <commit-sync/>
        <commit-confirm/>
        <commit-check/>
            <commit-boot/>
            <commit-comment>This is the comment</commit-comment>
    </commit-context>
</junos-context>
```

Child element nodes of <junos-context>:

- <hostname> - System hostname, same value as the $hostname parameter.
- <product> - Junos device model, same value as the $product parameter.
- <localtime> - Time that the script was invoked, same format as the $localtime parameter.
- <localtime-iso> - Time that the script was invoked, same format as the $localtime-iso parameter.

- <script-type> - Indicates the script type: op, event, or commit.
- <pid> - Process ID number for the cscript process that is executing the script.
- <tty> - TTY of the user's session if applicable. This node is not present if there is no associated TTY.
- <chassis> - Either TX Matrix (lcc or scc), JCS (psd or rsd), or otherwise it is "others".
- <routing-engine-name> - The name of the RE that the script is running on.
- <re-master> - When present, this node indicates that the script is running on the master routing-engine.

Child element nodes of <user-context>, which is a child node of <junos-context>:

- <user> - Local user account name of the user that invoked the script.
- <class-name> - Local class of user that invoked the script. (A "j-" is generally prepended to the class name unless the user is root, so a script should be prepared to remove a leading "j-" if one is present).
- <uid> - UID of user that invoked the script.
- <login-name> - Login-name of the user that invoked the script. If RADIUS or TACACS are in use, then this is the name on the remote authentication server, otherwise, it is the same as the <user> node.

The <user-context> hierarchy provides a more reliable user recognition method than the **$user** parameter because it reports both the RADIUS/TACACS login-name as well as the local user account. For example, in the example below the RADIUS account is admin and it is mapped to the local account remote-super:

```
    <user-context>
    <user>remote-super</user>
    <class-name>j-super-user-local</class-name>
    <uid>2005</uid>
    <login-name>admin</login-name>
</user-context>
```

The <op-context> child node is unique to op scripts:

- <via-url> - When present, this node indicates that the op script was executed via "op url"

The <commit-context> child node is unique to commit scripts:

- <commit-sync> - When present, this node indicates that a "commit synchronize" was performed.
- <commit-confirm> - When present, this node indicates that a "commit confirmed" was performed.
- <commit-check> - When present, this node indicates that a "commit check" was performed.
- <commit-boot> - When present, this node indicates that this is the initial boot-up commit.
- <commit-comment> - When present, this node indicates that a comment was included with the commit. The comment is included as the text content of this element node.

The **$junos-context** variable references the <junos-context> node, so location paths to the children of <junos-context> can branch immediately off of **$junos-context**.

Code example:

```
    <output> $junos-context/routing-engine-name;
```

Output:

```
    re0
```

Code example:

```
    <output> $junos-context/script-type;
```

Output:

```
op
```

Code example:

```
<output> $junos-context/user-context/uid;
```

Output:

```
2001
```

Example

This op script example shows how data can be retrieved from **$junos-context** that is then used to retrieve related information.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";

import "../import/junos.xsl";

match / {
    <op-script-results> {

        /* Display user name */
        <output> "Executed by: " _ $junos-context/user-context/login-name;
        <output> "Login class: " _ $junos-context/user-context/class-name;

        /* Get Login info */
        var $users = jcs:invoke( "get-system-users-information" );
        <output> "You logged in at: " _
            $users/uptime-information/user-table/
            user-entry[ tty == substring-after( $junos-context/tty, "/dev/tty" ) ]/login-time;

        /* Get script memory info */
        var $process-rpc = <command> "show system processes extensive";
        var $processes = jcs:invoke( $process-rpc );
        var $lines = jcs:break-lines( $processes );
        /* Find the right line */
        for-each( $lines ) {
            /* Find the one that starts with the PID and ends with cscript */
            var $tokens = str:tokenize( normalize-space( . ) );
            if( $tokens[1] == $junos-context/pid && $tokens[last()] == "cscript" ) {
                /* Display the memory used */
                <output> "cscript PID " _ $junos-context/pid _ " mem = " _ $tokens[6];
            }
        }
    }
}
```

Output

```
Executed by: jnpr
Login class: j-super-user-local
You logged in at: 7:57PM
cscript PID 1543 mem = 7364K
```

$product

Minimum Version: Junos 8.2

Description

The **$product** global parameter is a string containing the model of the Junos device that the script is running on.

Example values:

```
ex4200-24t
m120
mx480
mx80-48t
srx210h
```

Example

This op script example displays the device model that the script is running on.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$product = " _ $product;
    }
}
```

Output

```
$product = srx210h
```

$script

Minimum Version: Junos 8.2

Description

The **$script** global parameter is a string containing the filename of the script that is being executed. Typically it is just the actual filename with no path information; however, when “op url” is used, the **$script** parameter’s value looks like this:

```
/var/tmp/tmp_opeSuN6Y/tmp-op-script.alI53U/script.slax
```

Example

This op script example displays the filename of the current script.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$script = " _ $script;
    }
}
```

Output

```
$script = script.slax
```

$user

Minimum Version: Junos 8.2

Description

The **$user** global parameter is a string containing the name of the user that executed the script in the case of an op script or event script, or it is the name of the committing user in the case of a commit script. When RADIUS or TACACS authentication is used, the **$user** parameter is the login-name rather than the locally configured user account used for authorization.

The **$junos-context** global variable, added in Junos 11.1, is a better source of information when RADIUS or TACACS are in use because it includes both the login-name as well as the local user account.

Example

This op script example displays the current user.

Code

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
    <op-script-results> {
        <output> "$user = " _ $user;
    }
}
```

Output

```
$user = jnpr
```

Appendices

Appendix A: Dates, Times, and Durations

The EXSLT Dates and Times functions (e.g., date:add(), date:date-time(), etc.) supported by SLAX work with dates, times, and durations expressed in various formats, which are specified in Part 2 of the XML Schema and described below. These formats are based on ISO 8601 (with a few deviations), and they differ from the formats used for both the $localtime and $localtime-iso SLAX parameters. The below table demonstrates multiple date and time values in the three different formats.

$localtime	$localtime-iso	xs:dateTime (ISO 8601)
Tue Sep 21 14:36:27 2010	2010-09-21 14:36:27 MST	2010-09-21T14:36:27-07:00
Sat May 1 00:00:09 2010	2010-05-01 00:00:09 EDT	2010-05-01T00:00:09-05:00
Fri Jan 1 23:30:08 2010	2010-01-01 23:30:08 UTC	2010-01-01T23:30:08Z
Tue Sep 21 22:38:08 2010	2010-09-21 22:38:08 UTC	2010-09-21T22:38:08Z

Not all functions support all of the available formats, but each of the formats described below is supported by at least one EXSLT Dates and Times function. Consult each function's description to learn which formats are supported by each specific function.

- xs:dateTime – A specific date and time
  - Format: YYYY-MM-DDThh:mm:ss (Optionally: YYYY-MM-DDThh:mm:ss.ssszzzzzz)
    - YYYY - year – must be four or more digits and can be preceded with a minus sign.
    - MM – month - must be two digits.
    - DD – day – must be two digits.
    - T – date and time separator.
    - hh – hour in 24-hour format – must be two digits.
    - mm – minute – must be two digits.
    - ss – second – must be two digits.
    - .sss – decimal second – optional but can be one or more digits.
    - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
      - Examples: +05:30, -03:00
  - Examples:
    - 2010-09-21T16:19:00
    - 2001-01-01T00:00:00Z
    - 2010-05-01T15:30:01.113-03:00
    - 2008-12-31T23:59:05+05:00
- xs:date – A specific date
  - Format: YYYY-MM-DD (Optionally: YYYY-MM-DDzzzzzz)

        - YYYY - year – must be four or more digits and can be preceded with a minus sign.
        - MM – month - must be two digits.
        - DD – day – must be two digits.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - 2010-09-21
        - 2001-01-01Z
        - 2010-05-01-03:00
        - 2008-12-31+05:00
- xs:gYearMonth – A specific year and month
    - Format: YYYY-MM (Optionally: YYYY-MMzzzzzz)
        - YYYY - year – must be four or more digits and can be preceded with a minus sign.
        - MM – month - must be two digits.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - 2010-09
        - 2001-01Z
        - 2010-05-03:00
        - 2008-12+05:00
- xs:gYear – A specific year
    - Format: YYYY (Optionally: YYYYzzzzzz)
        - YYYY - year – must be four or more digits and can be preceded with a minus sign.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - 2010
        - 2001Z
        - 2010-03:00
        - 2008+05:00
- xs:gMonth – A recurring month
    - Format: --MM-- (Optionally: --MM--zzzzzz)
        - MM – month - must be two digits.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.

        - Examples: +05:30, -03:00
    - Examples:
        - --01--
        - --04--Z
        - --10---03:00
        - --12--+05:00
    - Note that the format for xs:gMonth used by SLAX is not the format specified in the XML Schema, because while the XML Schema used "--MM--" originally it later changed the xs:gMonth format to "--MM" in order to better comply with ISO 8601.
- xs:gMonthDay – A recurring day of year
    - Format: --MM-DD (Optionally: --MM-DDzzzzzz)
        - MM – month - must be two digits.
        - DD – day – must be two digits.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - --01-01
        - --04-05Z
        - --10-31-03:00
        - --12-25+05:00
- xs:gDay – A recurring day of month
    - Format: ---DD (Optionally: ---DDzzzzzz)
        - DD – day – must be two digits.
        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - ---01
        - ---05Z
        - ---31-03:00
        - ---25+05:00
- xs:time – A recurring time of day
    - Format: hh:mm:ss (Optionally: hh:mm:ss.ssszzzzzz)
        - hh – hour in 24-hour format – must be two digits.
        - mm – minute – must be two digits.
        - ss – second – must be two digits.
        - .sss – decimal second – optional but can be one or more digits.

        - zzzzzz – time-zone – optional but can be Z for UTC, or in the format: (+/-)hh:mm.
            - Examples: +05:30, -03:00
    - Examples:
        - 00:00:00
        - 23:59:59Z
        - 12:30:45.1234Z
        - 11:15:20+03:00
- xs:duration – A duration of time
    - Format: PnYnMnDTnHnMnS
        - P – Duration designator: must always be present.
        - nY – Number of years: optional.
        - nM – (Before T) Number of months: optional.
        - nD – Number of days: optional.
        - T – time designator: must be present if and only if one or more time values are included.
        - nH – Number of hours: optional.
        - nM – (After T) Number of minutes: optional.
        - nS – Number of seconds: optional and can include decimal digits.
    - Negative durations can be specified by including a minus before the P.
    - Examples:
        - P1Y
            - 1 year
        - P1Y2M3DT4H5M6S
            - 1 year, 2 months, 3 days, 4 hours, 5 minutes, and six seconds
        - P3D
            - 3 days
        - PT5M
            - 5 minutes

Appendix B: Qualified Names

Many of the names used within SLAX scripts, such as variable names and template names are referred to as qualified names, which are the combination of a namespace prefix and a local name, separated by a colon, or if no namespace prefix is assigned then the qualified name consists of simply the local name.

Example: "jcs:load-configuration" is a qualified name. "jcs" is the namespace prefix and "load-configuration" is the local name. A template name of "to-upper" could also be a qualified name, consisting of the local name of "to-upper" with no namespace prefix.

There are restrictions on the characters that are allowed as part of a qualified name, as well as the characters that are allowed as the first character. These restrictions apply to the namespace prefix and to the local name separately. In other words, both the namespace prefix and the local name have a first character that must abide by these restrictions:

- Permitted first character:
  - Letters: A-Z, a-z
  - Underscore: "_"
- Permitted characters:
  - Letters: A-Z, a-z
  - Digits: 0-9
  - Underscore: "_"
  - Hyphen: "-"
  - Dot: "."

The qualified name character restriction does not apply to the leading "$" in variable and parameter names; instead, the first character of the qualified name is considered to be the character following the "$".

At the time of this writing, the namespace prefix cannot begin with an underscore. Only the local name can begin with an underscore.

The character restrictions given above only consider ASCII characters, but there are wide ranges of non-ASCII Unicode characters that can be used within qualified names as well; however, the SLAX to XSLT conversion process does not currently handle the validation of non-ASCII characters consistently, so no attempt to document these ranges is made here; instead, the general advice provided is that only ASCII characters work reliably throughout all types of qualified names in SLAX.

For those interested in understanding what non-ASCII Unicode characters are currently permitted within qualified names, there are three different behaviors seen with the current SLAT to XSLT conversion process:

- Variables and parameters names cannot contain any non-ASCII characters.
- Qualified names that are not quoted (e.g. template names) allow most/all of the non-ASCII characters permitted by the XSLT 1.0 standard for qualified names, except non-ASCII characters are not allowed as the first character of the name.
- Qualified names that are quoted (e.g. the name attribute of **<xsl:element>**) allow most/all of the non-ASCII characters permitted by the XSLT 1.0 standard for qualified names, including the first character of the name.

Appendix C: XPath Axes, Node Tests, and Abbreviations

XPath is the language used by SLAX to retrieve information from XML documents. This Appendix includes tables that list the different XPath axes, node tests, and abbreviations.

XPath Axes:

Name	Description
`ancestor`	Contains all ancestor nodes of the context node. (The node's parent, the parent of its parent, etc.).
`ancestor-or-self`	Contains all ancestor nodes of the context node as well as the context node.
`attribute`	Contains all the attribute nodes of the context node.
`child`	Contains all the child nodes of the context node.
`descendant`	Contains all the descendant nodes of the context node. (The node's children, the children of its children, etc.).
`descendant-or-self`	Contains all the descendant nodes of the context node as well as the context node.
`following`	Contains all nodes that follow the context node in the XML document, except descendants, attributes, and comments.
`following-sibling`	Contains all following sibling nodes of the context node. Does not contain any nodes when the context node is an attribute or namespace node.
`namespace`	Contains the namespace nodes of the context node. Does not contain any nodes if the context node is not an element node.
`parent`	Contains the parent node of the context node, if one exists.
`preceding`	Contains all nodes that precede the context node in the XML document except for ancestors, attributes, and namespace nodes.
`preceding-sibling`	Contains the preceding siblings of the context node. Does not contain any nodes when the context node is an attribute or namespace node.
`self`	Contains the context node.

XPath Node Tests:

Node Test	Description
`qualified-name`	Matches nodes of the principal node type with the specified qualified name.
`*`	Matches nodes of the principal node type.
`prefix:*`	Matches nodes of the principal node type that have the specified namespace.
`comment()`	Matches comment nodes.
`node()`	Matches all nodes.

`processing-instruction()`	Matches processing instruction nodes.
`processing-instruction( "name" )`	Matches processing instruction nodes with the specified name.
`text()`	Matches text nodes.

XPath Abbreviations:

Abbreviation	Description
`(No axis specified)`	When no axis is specified, the default axis is the child axis.
`.`	(dot) Abbreviation for self::node() (Selects the context node)
`..`	Abbreviation for parent::node()
`@`	Abbreviation for the attribute axis
`//`	Abbreviation for /descendant-or-self::node()/