

Chapter 2

About FrameSLT XPath

FrameSLT supports a subset of the W3C XPath standard. Supported components should behave exactly to standard. Use of non-supported components will likely cause parsing errors or unexpected query results.

To use FrameSLT effectively, you must have a good working knowledge of XPath. You should review this information thoroughly before using FrameSLT, especially the details on which XPath components are supported and which are not. Nearly all FrameSLT functions rely on XPath to navigate the FrameMaker structure tree.

Expansion of the FrameSLT-supported XPath is dependent on the needs of users like you. If you have a need for an XPath component that is currently not supported, we'd like to hear from you at info@weststreetconsulting.com.

About XPath

The XPath specification, defined by the W3C Consortium, allows querying and navigation within an XML-style structure tree. It is sometimes considered a simple language in itself and is frequently used during XML transformations to query source documents for content. Unlike a "linear" search, XPath allows you to find elements and attributes under very specific conditions, including considerations of structural hierarchy, positioning, and node content.

XPath is ideal for navigating a FrameMaker structure tree, because the markup of such a tree is very much analogous to XML markup. Without a language such as XPath, you would be limited to basic name and content searches provided by the standard FrameMaker Find tool.

There are a wealth of resources available for learning XPath, including the W3C website at www.w3.org and free tutorials at websites such as www.w3schools.com. Because so many options are available, this document does not attempt to reproduce a complete XPath reference here. However, you can get some beginners tips with "*XPath quick primer*" on page 13. And, you can see plenty of samples in "*FrameSLT XPath examples*" on page 27.

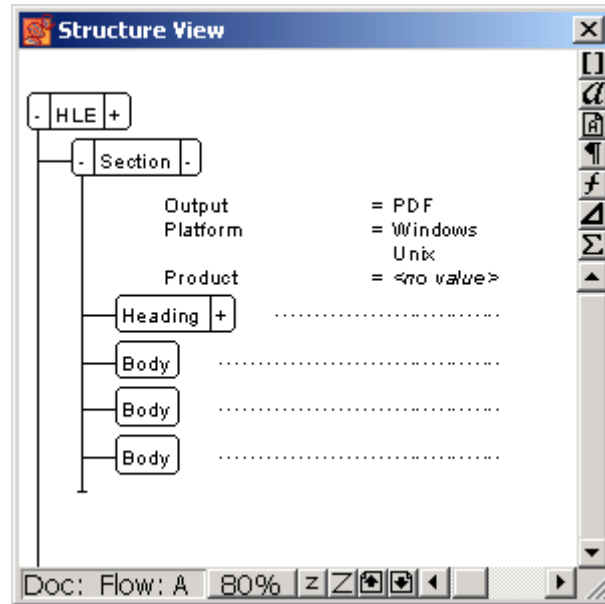
XPath quick primer

XPath is a special syntax designed for the express purpose of walking through a structure tree and finding very specific instances of elements, attributes, and other "nodes." It is reasonably simple to understand once you get started.

A node-matching expression is always a series of "axes" and "node tests." In essence, an axis tells which way to go, and the node test tells what to look for when you get there. For example, consider the following simple XPath:

`child::Body`

This expression says literally, "start at the context node (like an element), look to its children, and find any `Body` elements." Consider the following structure tree:



If the context element were the `Section` element, that XPath would find its three `Body` children. If the context were any other element, nothing would be found. In the FrameSLT Node Wizard, the currently selected element becomes the default context node. However, the selected element may not be relevant, if the first axis is a “go-to-root” axis, as explained in the next paragraph.

An important aspect of XPath is the first axis. In the previous example, the first axis (and only axis) is `child::` (go-to-child). So, a starting context must be manually provided (i.e., for the Node Wizard, the currently-selected element.) However, in many cases, especially with FrameSLT, you may find yourself using XPath that begins with the special “go-to-root” axis, indicated by a forward slash (/). This axis instructs the parser to begin at the root of the structure tree, using it as the initial context. With this axis, the context always starts at the root, and the currently-selected element is irrelevant.

As an example, the following XPath will find the highest-level element, `HLE`:

```
/child::HLE
```

It is very important to note that the forward slash *does not* set the `HLE` as the context... the context is actually “above” the `HLE`, at the true “root.” For example, the following XPath will find nothing, because the only child of the root is the highest level element, `HLE`:

```
/child::Section
```

However, the following expressions will find the `Section` element:

```
/child::HLE/child::Section
```

```
/descendant::Section
```

The descendant axis works because the `Section` is a descendant of the root. In fact, you can find any element by name with that particular expression. Note that the forward slash only means “go-to-root” if it is at the beginning. Otherwise, it is the delimiter between axis/node test components.

XPath also allows “predicates,” which are subexpressions in brackets used for testing something. You can use any axis in a predicate, and nest predicates within predicates as needed. For example, the following XPath will find the `Section` element again, because the predicate tests for the presence of an `Output` attribute:

```
/descendant::Section[attribute::Output]
```

In this case, the predicate doesn’t care what the value of `Output` is... only that the attribute exists. However, you can test values too, for example:

```
/descendant::Section[attribute::Platform = "Unix"]
```

That expression will find the `Section`, because the node test (`Section`) matches, and the predicate is satisfied. However, the following expression will find nothing, because the predicate is never satisfied:

```
/descendant::Section[attribute::Platform = "PDF"]
```

Once this begins to make sense, take a look at the examples in *“FrameSLT XPath examples”* on page 27. Before long, you should be able to master XPath, and see just how versatile and powerful it is as a structure query tool.

Nodes vs. elements—Terminology

When discussing XML and XPath, the word “node” is used frequently to describe a generic location type within a structure tree. A node can be a place such as an element, an attribute, or a namespace... essentially any definable place in the structure tree that a query can step to. As you study XPath elsewhere, you will find this word used much more frequently than “element” and “attribute.”

The FrameMaker interface and documentation, though, do not use this word, referring to locations specifically as elements and attributes. Therefore, the FrameSLT interface and documentation attempt to maintain this convention. However, when working with XPath, the word “node” is sometimes impossible to avoid, especially when the type of node is not specific. Therefore, an effort has been made in this document to adhere to the following terminology conventions:

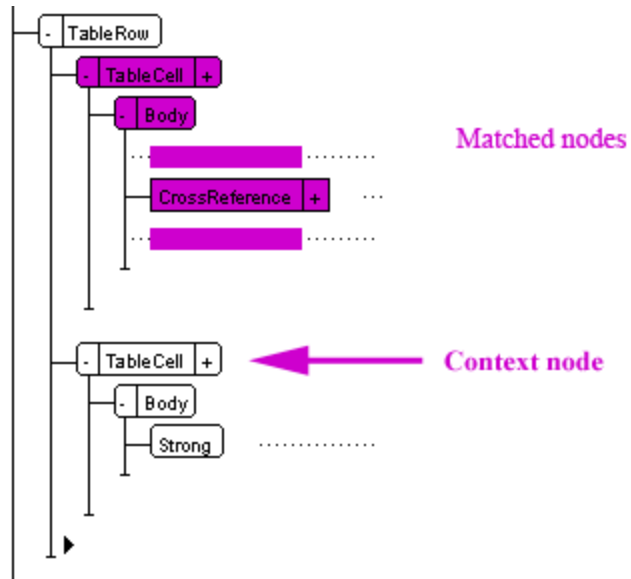
- **Node** When used alone, this word generally means “an element or attribute.”
- **Element node** A FrameMaker element
- **Attribute node** A FrameMaker attribute

In reality, the term “node” refers more generally to any point within a branching structure where branches begin, terminate, or propagate. For the purposes of this document, however, an association with elements and attributes should be sufficient.

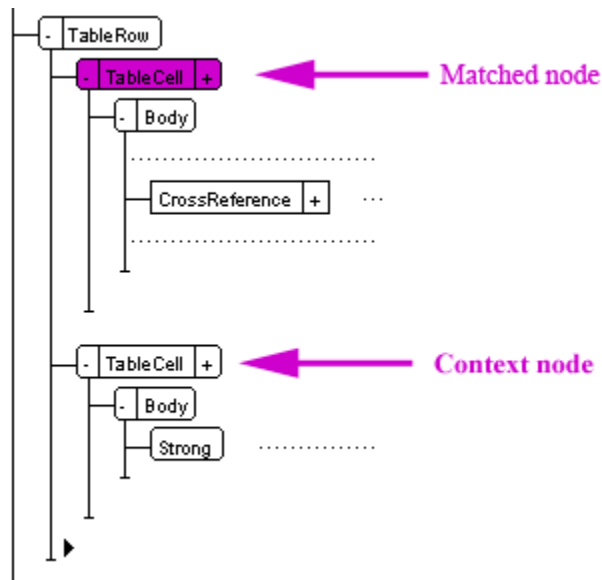
Supported axes

FrameSLT supports all standard XPath axes except `namespace::`. Using the “wildcard” character to indicate “any non-text node,” the following examples illustrate supported axes:

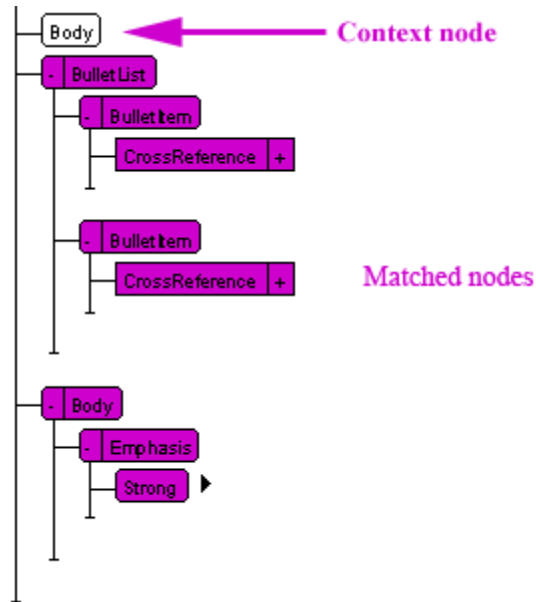
- `attribute::*`—Matches all attributes of the context node.
- `self::*`—Matches the context node.
- `child::*`—Matches all children of the context node.
- `descendant::*`—Matches all descendants (children, grandchildren, etc.) of the context node.
- `descendant-or-self::*`—Matches all descendants (children, grandchildren, etc.) of the context node, including the context node.
- `parent::*`—Matches the parent of the context node.
- `ancestor::*`—Matches all ancestors (parents, grandparents, etc.) of the context node
- `ancestor-or-self::*`—Matches all ancestors (parents, grandparents, etc.) of the context node, including the context node.
- `preceding::*`—Matches all preceding sibling nodes and all descendants of them, in document order. For example:



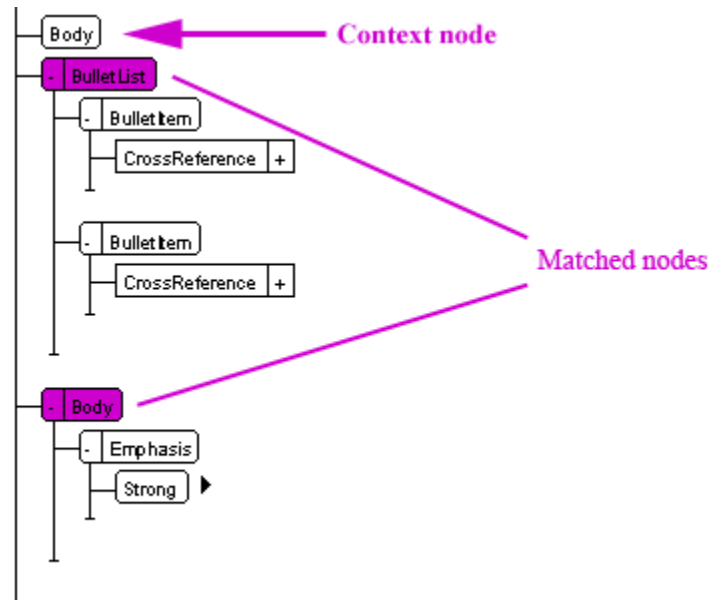
- `preceding-sibling::*`—Matches all preceding sibling nodes only, and excludes descendants, in document order. For example:



- `following::*`—Matches all following sibling nodes and all descendants of them, in document order. For example:



- `following-sibling::*`—Matches all following sibling nodes only, and excludes descendants, in document order. For example:



Special “fmp” axis

FrameSLT implements a non-standard `fmp` axis for querying FrameMaker-specific object properties. The notation is similar to standard W3C-defined axes, but rather than indicating movement towards a node in the structure tree, it directs the retrieval of some property associated with the current element node (that is, the context node).

As an example, the following expression will match all elements that have the “Body” paragraph format applied to the underlying paragraph, or the first underlying paragraph if the element wraps multiple paragraphs:

```
//*[fmp:Pgftag="Body"]
```

...where `PgfTag` is the specific notation that indicates a paragraph tag query. As another example, assuming that `Graphic` is a graphic element, the following expression matches all `Graphic` elements whose underlying anchored frame contains a referenced PNG file:

```
//Graphic[contains(fmprop::ImportObFile, ".png")]
```

Note that this evaluation would be case-sensitive, so a file with a `.PNG` extension would not make a match, unless the non-standard `contains-ci()` function were used instead. For more information, see [“Supported functions”](#) on page 23.

Currently, a very small subset of FrameMaker properties is supported by the `fmprop` axis, as described in the following table. There are many hundreds of potential properties available for evaluation, so it is not feasible to implement all of them at once. However, new properties will be added upon request. If you have a need to query a certain type of property, please contact us and we may be able to issue you a patch.

Additionally, note the following:

- The syntax of these properties follows the MIF tag format.
- Most of these properties can also be set using Node Wizard scripts.

fmprop property	What is retrieved
ImportObFile	Full path of each imported (referenced) file in the underlying anchored frame. If the test matches a single file, the predicate is considered satisfied. This property is relevant to graphic elements only. Example: <pre>//Graphic[contains-ci(fmprop::ImportObFile, ".png")]</pre>
PgfTag	Paragraph tag assigned to the span of text that the element wraps. Example: <pre>//*[fmprop::PgfTag="Body"]</pre>
TblTag	Table format tag of the current table, only applicable for table component elements. Any elements that are not table components are automatically disqualified by this test. Do not use this property to test paragraph container elements inside table cells; rather, use the <code>ancestor</code> axis to test the ancestor cell, row, or table instead. Example: <pre>//Table[fmprop::TblTag="Ruling"]</pre> In the previous example, if <code>Table</code> elements are not table components, the expression will never match anything, regardless of the text within the quotation marks:
FChangeBar	Change bar status, either “true” or “false”. It can be used to find elements whose first paragraph (or parent paragraph, for text-range elements) is marked with a change bar. This setting is Boolean in nature and is only applicable with the “true” and “false” arguments. Examples: <pre>//p[fmprop::FChangeBar="true"]</pre> <pre>//title[fmprop::FChangeBar="false"]</pre>

fmprop property

XRefName
XRefSrcText
XRefSrcFile

What is retrieved

Cross-reference-related properties of the underlying cross-reference object, only applicable for cross-reference elements. Any elements that are not cross-references are automatically disqualified by these tests. Further descriptions are as follows:

- **XRefName** - The cross-reference format.
- **XRefSrcText** - The reference ID; that is, the value of the “IDReference” attribute of cross-reference element.
- **XRefSrcFile** - The source file that contains the destination of the cross-reference. If the cross-reference is internal to the document, the query returns an empty string.

The following example matches all `xref` elements that use the “Heading on page” format and have destinations within the current document:

```
//xref[fmprop:XRefName="Heading on page" and
fmprop:XRefSrcFile=""]
```

The following example matches all `xref` elements whose destination is located in the external file “somefile.fm”:

```
//xref[contains(fmprop:XRefSrcFile,"somefile.fm")]
```

In both examples, if the `xref` element is not a cross-reference element, the expressions would never match anything, regardless of the text within the quoted strings.

MTypeName
MText

Marker-related properties of the underlying marker object, only applicable for marker elements. Any elements that are not markers are automatically disqualified by these tests. Further descriptions are as follows:

- **MTypeName** - The marker type
- **MText** - The marker text

The following example matches all `IndexMarker` elements that use the “Index” type and contain the text “XPath”:

```
//IndexMarker[fmprop:MTypeName="Index" and
contains(fmprop:MText,"XPath")]
```

In this example, if the `IndexMarker` element is not a marker element, the expression would never match anything, regardless of the text within the quoted strings.

Special book- and file-related axes

FrameSLT implements the following non-standard axes that allow a query to traverse from documents to books and vice-versa, and directly from one file to another. For example, if you want to perform operations on a whole book using the Node Wizard or Node Wizard scripts, you would need to use one or more of these axes.

Note: The behavior of these axes can be difficult to understand. However, they are very important for advanced FrameSLT usage. If you need assistance with expression syntax, please contact West Street. For extended examples, see [“Cross-book and](#)

cross-file queries” on page 30.

Axis	Behavior
<code>fmbook</code>	<p>Matches:</p> <ul style="list-style-type: none"> • The highest-level element (HLE) of the active book -or- • If no book is active, the HLE of the first book that can be associated with the active document -or- • Nothing, if no corresponding book can be found or no document is active at all <p>This axis is the primary workhorse for stepping from a document tree into a book structure tree, noting that it goes straight to the book HLE and does not consider any current context. Element names are currently not considered, so the node test should always be simply an asterisk (*). For example, the following simple expression is valid and matches a book HLE:</p> <p><code>fmbook::*</code></p> <p>This expression will match the book HLE if the book is active or any of its chapters are active. Again, note that the current element selection or insertion point location is not relevant.</p>
<code>fmcomp</code>	<p>Matches the component-level element in book structure tree for the currently-active document. That is, it searches for an open book that contains the currently-active document as a chapter, then matches the respective component element in book structure tree. If the context is already a book structure tree, it matches nothing.</p> <p>This axis is an alternative for moving from a document structure tree to a book. In most cases, <code>fmbook::*</code> may be more appropriate. Note that:</p> <ul style="list-style-type: none"> • This axis does not consider element names; therefore, the node test should always be an asterisk (*) • Like <code>fmbook::*</code>, the axis will match the component element regardless of the current context in the document. That is, the context does not need to be the document HLE.

Axis	Behavior
fmchap	<p>Matches the HLE of the document associated with the current book component element; that is, the corresponding chapter file. It only matches if:</p> <ul style="list-style-type: none"> • The current context is a component-level element within a book structure tree • The associated chapter is currently open, unless the expression is being used by a feature that also supports automatic file opening, such as Node Wizard scripts <p>For example, assuming that a book is active, the following expression will match the HLEs of all chapter documents of the book:</p> <pre>//*/fmchap::*</pre> <p>If no book is active, the expression would match nothing. Note the following:</p> <ul style="list-style-type: none"> • This axis does not consider element names; therefore, the node test should always be an asterisk (*). • This axis matches HLEs in the main flow only. You cannot step from a book into any flow other than the main flow.
fmfile	<p>Matches the HLE of the file specified as the node test. You can specify:</p> <ul style="list-style-type: none"> • An absolute path • A relative path (relative the currently-active file) • The filename of any open file, regardless of its actual location in the file system <p>For example, the following expression matches the HLE of the file <code>somefile.fm</code>:</p> <pre>fmfile::somefile.fm</pre> <p>Note the following:</p> <ul style="list-style-type: none"> • The axis is valid for both document and book files. For document files, the axis will match the HLE of the main flow only, not any other flow. • If path separators are required, use forward slashes, for example: <pre>fmfile::C:/MyDocs/somefile.fm</pre> • If the path contains any whitespace, you must enclose it in quotes, for example: <pre>fmfile::"C:/My Docs/some file.fm"</pre> • The target file must be currently open, unless the feature using the expression provides file-opening capabilities, such as Node Wizard scripts;

The following example matches all `Body` elements in an entire book, regardless of whether the book or a chapter file is currently active. It includes a diagram of how the axes are working. For more examples, see [“Cross-book and cross-file queries”](#) on page 30.

fmbook – Matches the book HLE.

fmchap – For any elements matched by the previous axis that happen to be component-level elements, matches the HLE of the corresponding chapter (document) file.

fmbook::*/ */ fmchap::*/ */ Body

Starting from the context of the book HLE, matches all descendant-or-self elements. That is, all elements in the book structure tree.

Starting from the context of the document HLE, matches all descendant-or-self elements that are **Body** elements. The behavior is identical to a scenario where you had the document HLE selected, then issued:

descendant-or-self::Body

Abbreviated axes

FrameSLT supports most XPath abbreviations for supported axes and functions, as shown in the following examples. If not shown, the abbreviation is not supported.

Abbreviated syntax

```
/Section/Para
/Section[@Output = "PDF"]
//Section/Para
.
..
/Section[5]
/Body[last()]
```

Equivalent long version

```
/child::Section/child::Para
Section[attribute::Output = "PDF"]
/descendant-or-self::Section/child::Para
self::node()
parent::node()
/child::Section[position() = 5]
child::Body[position() = last()]
```

Supported logical test operators

Operator	Meaning
= or ==	equals
!=	does not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Examples:

child::Para[position() >= 5] Select all **Para** children in the fifth position or higher.

Heading[. != "This is a heading"] Select all **Heading** children that *do not* contain the text "This is a heading."

Conditional[@Output = "PDF"] Select all **Conditional** children that have an **Output** attribute, and at least one of the values is **PDF**.

Supported functions

FrameSLT XPath supports the following functions:

- `position()`
- `last()`
- `contains()`
- `contains-ci()`
- `starts-with()`
- `starts-with-ci()`
- `not()`

The following sections describe these functions in more detail.

Node position functions

FrameSLT supports the following position-related functions:

- **position()** Returns an element node's position in a branch relative to its siblings. The behavior of this function differs according to the most recent previous axis. See the W3C documentation for more information.
- **last()** Returns the position of the last element node in the branch containing the context element node

For example, a test for `position() = 3` would only match if the element were in the third position. Or, a test for `last() = 3` would only match if the element were on last on a branch and in the third position.

In an expression, the order of functions and operational terms is unimportant. For example, `position() = 3` means the same as `3 = position()`.

For more detailed examples, see ["FrameSLT XPath examples"](#) on page 27.

Node content functions

FrameSLT supports the following content-related functions:

- **contains(x,y)** Returns the string "true" if the string "x" contains the string "y", otherwise returns the string "false". This function *is* case-sensitive.
- **contains-ci(x,y)** Returns the string "true" if the string "x" contains the string "y", otherwise returns the string "false". This function *is not* case-sensitive.
- **starts-with(x,y)** Returns the string "true" if the string "x" starts with the string "y", otherwise returns the string "false". This function *is* case-sensitive.
- **starts-with-ci(x,y)** Returns the string "true" if the string "x" starts with the string "y", otherwise returns the string "false". This function *is not* case-sensitive.

Note: `contains-ci()` and `starts-with-ci()` are not part of the W3C XPath recommendation. They are "add-on" functions provided with FrameSLT for your convenience.

All of these functions require two arguments, which can either be a literal string or a node test. In the case of a node test, the content of the matched node becomes the string for comparison when the function is evaluated. If any node test for any argument fails, the function will return "false."

As an example, the following function will return "true" if a child **Heading** element contains the text "mytext":

```
contains(Heading,"mytext")
```

The following function will return “true” if the current context node contains this text:

```
contains(., "mytext")
```

Functions such as these are used in predicates, and if a string comparison operator is missing, the parser assumes a match of “true” to satisfy the predicate. Therefore, the following XPath expressions are functionally equivalent:

```
//*[contains(., "mytext")]
//*[contains(., "mytext") = "true"]
//*[contains(., "mytext") != "false"]
```

These expressions will all match any element in the tree that contains the text string “mytext”.

For more detailed examples, see [“FrameSLT XPath examples”](#) on page 27.

Boolean functions

FrameSLT supports the following Boolean-related function:

not() Returns either the string “true” or “false”, intending to represent the opposite of the return of its argument.

`not()` always takes a single argument. If the argument is a node test (that is, returns a node value), the function will return “false” if a node is found, otherwise it returns “true”. For example, the following function will return “true” only if a child `Heading` element does not exist, with respect to the current context:

```
not(Heading)
```

Or as another example, the following function will return true only if the context element itself is not named `Heading`:

```
not(self::Heading)
```

If the argument returns a string value, `not()` will return “true” only if the return string is empty or equals “false”. For example, the following functions will return “true”:

```
not("")
not("false")
```

Besides literal strings, any argument that returns a literal string, such as another function, is evaluated in the same fashion. For example, the following function will return “true” only if the context node does not contain the text “mytext”:

```
not(contains(., "mytext"))
```

The `not()` function is a powerful tool that can make XPath queries more precise, but the logic can quickly become complex. For more detailed examples, see [“FrameSLT XPath examples”](#) on page 27.

Node test wildcards

FrameSLT supports the asterisk (*) wildcard for node testing, which indicates “any” element node. For example, the following expression will match every element in the document:

```
//*
```

The asterisk will not match text nodes, and it must appear alone. For example, you cannot use:

```
//B*dy
```

...to match `Body` elements.

EDD-applied prefixes/suffixes and node testing

When you test an element for content, such as in the following expression:

```
//Section[Heading = "My Heading"]
```

...no prefixes or suffixes applied by the EDD are considered. Therefore, in the example above, the `Heading` element would have to contain the text “My Heading” as typed by an author, and any EDD prefixes and/or suffixes are completely ignored.

Unsupported syntax

The following types of XPath syntax are not supported by FrameSLT:

Parentetical expressions in compound logical tests

Compound logical tests are supported, but not with parentetical expressions. Therefore, compound conjunctions are also not supported. For example, the following expression cannot be processed:

```
Body[. = "MyText" and (last() or 5)]
```

Because “back-to-back” predicates are considered to have an “and” logic, the following expression is also not supported:

```
Body[. = "MyText"][5 or last()]
```

However, all of these situations can be replicated in a longer form, using the “self” axis and multiple predicates, for example:

```
Body[. = "MyText" and .[last() or 5]]
```

Abbreviated attribute and value test

The following abbreviated syntax for testing an attribute value is not supported:

```
Body[@Output ("PDF")]
```

Instead, use the following:

```
Body[@Output = "PDF"]
```

Standalone “go-to-root” XPath expressions

The following expression has no relevance in FrameSLT and is therefore not supported:

```
/
```

With XSLT, you might see this XPath expression frequently in `template` elements, such as `<xsl:template match="/">`. However, this concept has no application in FrameSLT and therefore the expression cannot be parsed.

Direct syntax to unique ID attribute nodes

The following syntax, used to select an element node with a particular unique ID attribute, is not supported:

```
ElementName ("ID")
```

For example, the following expression, used to find a child `Body` element with the “MyID” unique ID, cannot be parsed:

```
Body ("MyID")
```

If you require a query using a unique ID attribute, use the attribute name directly. For example:

```
Body[@ID = "MyID"]
```

Limitations and known issues

The following sections describe known discrepancies between the established XPath standard and FrameSLT XPath.

Testing element node text

When testing the text of an element node, only the first paragraph is tested. This includes expressions with whole string evaluations and expressions with functions such as:

```
//Section[Heading = "MyHeading"]
//Body[contains(., "some text")]
//BulletList[starts-with(., "R")]
```

This limitation is set because test strings could otherwise become enormously long, such as testing the text of the highest-level element of a 200 page document. Strings of this length would adversely affect performance and likely cause crashes. If you need to test the text in a higher-level element, consider using predicates to test subordinate elements, accomplishing the same goal while reducing the processing strain. For example, instead of:

```
//Section[contains(., "some text")]
```

...you could use an expression such as:

```
//Section[descendant::*[contains(., "some text")]]
```

or the following equivalent expression:

```
//Section[contains(*, "some text")]
```

This limitation does not apply to testing attribute values. For attribute nodes, all text of all values is always tested.

Finding text() nodes with no siblings

All elements that contain text also have an implied text node, the text itself. While FrameSLT supports the text() node test, it will not find any text nodes that have no siblings. That is, if a text() node has no element node siblings, FrameSLT XPath is currently unable to find it.

It is hoped that this issue should rarely be of importance in FrameSLT functionality, because FrameMaker's internal representation of structure would make it difficult to support such XPath constructions. For Node Wizard functions, you can use actions such as "Wrap contents in" and "Paste clipboard over contents" to work around the issue.

Comparing two nodes without a bracketed predicate

In most cases, FrameSLT supports the shorthand syntax for testing node content, such as:

```
//Heading = "My Heading"
```

...which is equivalent to:

```
//Heading[. = "My Heading"]
```

The shorter version will not work, however, if you are attempting to compare two node sets. For example, the following expression is not supported:

```
//Heading = Body
```

To accomplish this type of query, you must write it out with an explicit bracketed predicate using a "to self" node, such as:

```
//Heading[. = Body]
```

Normally, these types of comparisons are rare. Note that this limitation applies to the "baseline" expression only. If the test is already within a predicate, the workaround is not necessary. For example, the following expression will work fine:

```
//Section[Heading = Body]
```

In some cases with long, complex expressions, the shorthand format has exhibited problems. In these rare cases, the longer format can be used to work around the bug.

FrameSLT XPath examples

Tips: Always enclose all string literals in single or double quotes. If your literal must contain double quotes itself, enclose the literal in single quotes, and vice-versa.

Do not enclose integers in quotes.

Don't forget the parenthesis on functions, such as `position()`. Without the parenthesis, FrameSLT will think it is simply looking for an element named `position`.

Remember that XPath expressions can become long and complex. Any error, even as small as a single character, will likely cause an expression to fail.

"Single document" queries

The following examples are applicable for querying within a single document; that is, no traversing across books or between separate files.

Expression	Meaning
<code>Body</code>	Match all <code>Body</code> children of the context node.
<code>/Body</code>	Match all <code>Body</code> children of the highest-level element (HLE)
<code>Body[1]</code>	Match the first <code>Body</code> child of the context node.
<code>//Body</code>	Match all <code>Body</code> descendants of the HLE, and the HLE if it is a <code>Body</code> .
<code>/descendant::Body[1]</code>	Match all <code>Body</code> descendants of the HLE, that are the first <code>Body</code> elements in their respective branches.
<code>Chapter/Section//Body</code>	Match all <code>Body</code> descendants of the <code>Section</code> children of <code>Chapter</code>
<code>Chapter/Section//text()</code>	Match all text node descendants of the <code>Section</code> children of <code>Chapter</code>
<code>Body/parent::Section</code>	Match all <code>Body</code> elements with a <code>Section</code> parent
<code>Body/ancestor::Section</code>	Match all <code>Body</code> elements with a <code>Section</code> parent or at least one <code>Section</code> ancestor
<code>Body/ancestor::Section/ancestor::Section</code>	Match all <code>Body</code> elements with at least two <code>Section</code> ancestors
<code>../Body</code>	Match all <code>Body</code> siblings of the context node
<code>/*</code>	Match the highest-level element.
<code>//node()</code>	Match every element and text node in the tree.
<code>//*</code>	Match every element node in the tree.
<code>//text()</code>	Match every text node in the tree.
<code>//*[@Output]</code>	Match every element node in the tree with an <code>Output</code> attribute

Expression

```

/*[@Output = "PDF"]

/*[@Output != "PDF" or @Output != ""]

/*[@*]

//Body[../@Output = "PDF"]
    or
//Body[../@Output = "PDF"]
//Body[parent::Section/@Output = "PDF"]
    or
//Body[parent::Section[@Output = "PDF"]]
//Body[parent::Section[3]]

//Body[last() = 5]

/*[5 and 4]

/*[@Output = "PDF"][5]

//Section[Heading = "This text"]

/*[position() > 3 or 5 > position()]

//Heading[. = "This text"]
    or
//Heading = "This text"
//Heading[. > "MyHeading"]

```

Meaning

Match every element node in the tree with an `Output` attribute set to `PDF`. In FrameSLT, if the attribute has multiple values, they are all considered.

Match every element node in the tree with an `Output` attribute not set to `"PDF"` (any of the attribute's values), or not empty.

Match every element node in the tree that has at least one attribute, regardless of the attribute contents, if any.

Match every `Body` element in the tree whose parent has an `Output` attribute set to `"PDF"`.

Match every `Body` element in the tree with a `Section` parent, whose `Output` attribute is set to `"PDF"`.

Match every `Body` element that has a `Section` parent, which is third `Section` element on the branch.

Match every `Body` element in the tree that has exactly four `Body` element siblings.

Matches nothing. An element cannot occupy two positions.

Matches the same thing as:

```
/*[@Output = PDF and 5]
```

Match every `Section` element node in the tree with a `Heading` child, with the text `"This text"`.

Matches the same thing as:

```
/*[4]
```

Match every `Heading` element node with the text `"This text."`

Match every `Heading` element node with text alphabetically greater than `"MyHeading"`, such as a `Heading` with the text `"YourHeading."`

Note: This type of test is more appropriate for text strings with no spaces. If you attempt to alphabetically compare strings with multiple words, the results may not be as reliable.

Expression	Meaning
<code>//*[@Output = "PDF" or Body = "text" or 5 or 4 or last() or .]</code>	Match every element node in the tree. The final “to self” (.) test satisfies everything and negates all other logical tests if they fail.
<code>//*[contains(., "mytext")]</code>	Match every element node in the tree that contains the text “mytext”.
<code>//*[contains-ci(., "mytext")]</code>	Match every element node in the tree that contains the text “mytext”, without regard for case-sensitivity.
<code>//*[contains(@*, "MyValue")]</code>	Match every element node in the tree that has an attribute that contains the text “MyValue”.
<code>//*[not(contains(@*, "MyValue"))]</code>	Match every element node in the tree that does not have any attribute that contains the text “MyValue”.
<code>//Section[not(Body)]</code>	Match every <code>Section</code> element in the tree that does not have a child element named <code>Body</code> .
<code>//Section[not(Body[contains(&*, "MyValue")])]</code>	Match every <code>Section</code> element in the tree that does not have a child element named <code>Body</code> with any attribute containing the text “MyValue”.
<code>//*[not(self::*[position() = last()])]</code>	Match every element node that is not the last element in its respective branch.
<code>//Heading[starts-with(., "R")]</code>	Match every <code>Heading</code> element that starts with the letter “R”.
<code>//Heading[starts-with(., "R") or contains(., "My Heading")]</code>	Match every <code>Heading</code> element that starts with the letter “R” or contains the text “My Heading”.
<code>//Section[Heading = Body]</code>	Match every <code>Section</code> element that has a <code>Heading</code> child and a <code>Body</code> child that both contain exactly the same text.
<code>//Section[contains(Body, Heading)]</code>	Match every <code>Section</code> element that has any <code>Body</code> child that contains the whole text string wrapped in any <code>Heading</code> child.

Cross-book and cross-file queries

The following examples use the special axes for traversing between files and books, as described under “*Special book- and file-related axes*” on page 19.

Expression

`fmbook::*`

-Or-

`/fmbook::*`

`//fmbook::*`

`fmbook::*/*`

`fmbook::*//*/fmchap::*`

`fmbook::*//*/fmchap::*[self::Chapter]`

`fmbook::*//*/fmchap::*//Body`

Behavior

If the context is an active document:

Matches the highest-level element (HLE) of the “parent” book for the active document; that is, the first active book that can be found that contains the active document as a chapter. If no applicable book can be found, it matches nothing.

If the context is an active book:

Matches the HLE of the book.

Syntax error, because it effectively represents two axes back-to-back (“descendent-or-self” and “fmbook”).

If the context is an active document:

Matches all child elements of the “parent” book HLE. In a traditional book without folders and groups, it would match all component elements. If no applicable book can be found, it matches nothing.

If the context is an active book:

Matches all children of the book HLE.

If the context is an active document:

Matches the HLEs of all chapter documents in the parent book.

If the context is an active book:

Matches the HLEs of all chapter documents in the book.

If the context is an active document:

Matches the HLEs of all chapter documents in the parent book that have the tag `Chapter`.

If the context is an active book:

Matches the HLEs of all chapter documents in the book that have the tag `Chapter`.

If the context is an active document:

Matches all `Body` elements in the parent book.

If the context is an active book:

Matches all `Body` elements in the book.

Expression

`fmcomp::*//ancestor::*//*/fmchap::*//
Body`

`fmfile::somefile.fm`

`fmfile::somefile.fm//Body`

`fmfile::somefile.fm/
fmfile::someotherfile.fm/Body`

`fmfile::somefile.fm/fmbook::*//*/
fmchap::*//Body`

Behavior

If the context is an active document:

Matches all `Body` elements in the parent book.

If the context is an active book:

Matches nothing. `fmcomp` is only relevant when the context is a document.

In all contexts:

Matches the HLE of a file with the name `somefile.fm`. The remainder of the absolute path is not relevant, unless you are attempting to open the file, in which case the file must be in the same folder as the current context file.

In all contexts:

Matches all `Body` elements in the file `somefile.fm`.

In all contexts:

Matches all `Body` elements in the file `someotherfile.fm`, provided that an HLE for `somefile.fm` was found first. The expression effectively steps through multiple documents in a single query.

In all contexts:

Matches all `Body` elements in the parent book for the file `somefile.fm`.

