# XProc 3.0: Standard Step Library

## Community Group Report 12 September 2022

**Specification:**

https://spec.xproc.org/3.0/steps/

**Editors:**

Norman Walsh

Achim Berndzen

Gerrit Imsieke

Erik Siegel

**Participate:**

GitHub xproc/3.0-steps

Report an issue

**Errata:**

https://spec.xproc.org/3.0/steps/errata.html

This document is also available in these non-normative formats: XML, PDF (A4), PDF (US Letter).

### Abstract

This specification describes the standard step vocabulary of *XProc 3.0: An XML Pipeline Language*.

# Status of this Document

This specification was published by the XProc Next Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

If you wish to make comments regarding this document, please send them to xproc-dev@w3.org. (subscribe, archives).

This document is derived from XProc: An XML Pipeline Language published by the W3C.

# Table of Contents

# 1. Introduction

This specification describes the standard, required atomic XProc steps. A machine-readable description of these steps may be found in steps.xpl.

Many atomic steps are available for [XProc 3.0]. They are described in several specifications. This specification describes the general background common to all steps. A conformant processor **must** implement all of the steps in this specification. Additional steps may also be implemented.

The types given for options should be understood as follows:

- Types in the XML Schema namespace, identified as QNames with the `xs:` prefix, as per the XML Schema specification with one exception. Anywhere an `xs:QName` is specified, an EQName is allowed.
- `XPathExpression`: As a string per [W3C XML Schema: Part 2], including whitespace normalization, and the further requirement to be a conformant Expression per [XPath 3.1].
- `XSLTSelectionPattern`: As a string per [XSLT 3.0] conforming to an XSLT *selection pattern*.
- `XPathSequenceType`: An XPath sequence type.
- `ContentType`: A media type as defined in [RFC 2046].
- `ContentTypes`: As a whitespace separated list of media types as defined in [RFC 2046].

Option values are often expressed using the shortcut syntax. In these cases, the option shortcuts are generally treated as value templates. However, for options of type `map()` or `array()`, an expression is *required* (there is no non-expression string which can ever be a legal value for a map or array). Given that every value entered this way will have to be a value template, and consequently every curly brace contained within the expression will have to be escaped, values of type map or array are defined to be expressions directly.

Some aspects of documents are generally unchanged by steps:

- When a step in this library produces an output document, the base URI of the output is the base URI of the step's primary input document unless the step's process explicitly sets an `xml:base` attribute or the step's description explicitly states how the base URI is constructed.

- Steps are responsible for describing how document properties are transformed as documents flow through them. Many steps claim that the specified properties are preserved. Generally, it is the responsibility of the pipeline author to determine when this is inapropriate and take corrective action. However, it is the responsibility of the pipeline processor to assure that the `content-type` property is correct. If a step transforms a document in a manner that is inconsistent with the `content-type` property (accepting an XML document on the source port but producing a text document on the result, for example), the processor must assure that the `content-type` property is appropriate. If a step changes the `content-type` in this way, it **must** also remove the `serialization` property

Also, in this specification, several steps use this element for result information:

```
<c:result>
    string
</c:result>
```

When a step uses an XPath to compute an option value, the XPath context is as defined in [XProc 3.0].

When a step specifies a particular version of a technology, implementations **must** implement that version or a subsequent version that is backwards compatible with that version. At user-option, they may implement other non-backwards compatible versions.

In this specification the words **must**, **must not**, **should**, **should not**, **may** and **recommended** are to be interpreted as described in [RFC 2119].

As described in *PSVIs in XProc* in *XProc 3.0: An XML Pipeline Language*, steps may not produce PSVI output unless that behavior is explicitly described. In this specification, the steps that may produce PSVI output are the "identity" steps: `p:identity`, `p:store`, and `p:split-sequence` (which **must** preserve PSVI properties that appear

on their inputs). In addition, the `p:xslt` and `p:xquery` steps **may** return documents with PSVI annotations.

## 2. The required steps

A conformant processor must support all of these steps.

### 2.1. p:add-attribute

The `p:add-attribute` step adds a single attribute to a set of matching elements. The input document specified on the `source` is processed for matches specified by the *selection pattern* in the `match` option. For each of these matches, the attribute whose name is specified by the `attribute-name` option is set to the attribute value specified by the `attribute-value` option.

The resulting document is produced on the `result` output port and consists of a exact copy of the input with the exception of the matched elements. Each of the matched elements is copied to the output with the addition of the specified attribute with the specified value.

```
<p:declare-step type="p:add-attribute">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
  <p:option name="attribute-name" required="true" as="xs:QName"/>
  <p:option name="attribute-value" required="true" as="xs:string"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if the *selection pattern* matches a node which is not an element.

The value of the `attribute-value` option **must** be a legal attribute value according to XML.

If an attribute with the same name as the expanded name from the `attribute-name` option exists on the matched element, the value specified in the `attribute-value` option is used to set the value of that existing attribute. That is, the value of the existing attribute is changed to the `attribute-value` value.

> **Note**
>
> If multiple attributes need to be set on the same element(s), the `p:set-attributes` step can be used to set them all at once.

This step cannot be used to add namespace declarations. It is a *dynamic error* (`err:XC0059`) if the QName value in the `attribute-name` option uses the prefix "`xmlns`" or any other prefix that resolves to the namespace name `http://www.w3.org/2000/xmlns/`. Note, however, that while namespace declarations cannot be added explicitly by this step, adding an attribute whose name is in a namespace for which there is no namespace declaration in scope on the matched element may result in a namespace binding being added by namespace fixup.

If an attribute named `xml:base` is added or changed, the base URI of the element **must** also be amended accordingly.

#### Document properties

All document properties are preserved.

## 2.2. p:add-xml-base

The `p:add-xml-base` step exposes the base URI via explicit `xml:base` attributes. The input document from the `source` port is replicated to the `result` port with `xml:base` attributes added to or corrected on each element as specified by the options on this step.

```
<p:declare-step type="p:add-xml-base">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="all" as="xs:boolean" select="false()"/>
  <p:option name="relative" as="xs:boolean" select="true()"/>
</p:declare-step>
```

The value of the `all` option **must** be a boolean.

The value of the `relative` option **must** be a boolean.

It is a *dynamic error* (`err:XC0058`) if the `all` and `relative` options are *both* `true`.

The `p:add-xml-base` step modifies its input as follows:

- For every element that is a child of the document node: force the element to have an `xml:base` attribute with the document's [base URI] property's value as its value.
- For other elements:
  - If the `all` option has the value `true`, force the element to have an `xml:base` attribute with the element's [base URI] value as its value.
  - If the element's [base URI] is different from the its parent's [base URI], force the element to have an `xml:base` attribute with the following value: if the value of the `relative` option is `true`, a string which, when resolved against the parent's [base URI], will give the element's [base URI], otherwise the element's [base URI].
  - Otherwise, if there is an `xml:base` attribute present, remove it.

**Document properties**

All document properties are preserved.

## 2.3. p:archive

The `p:archive` step outputs on its `result` port an archive (usually binary) document, for instance a ZIP file. A specification of the contents of the archive may be specified in a manifest XML document on the `manifest` port. The step produces a report on the `report` port, which contains the manifest, amended with additional information about the archiving.

```
<p:declare-step type="p:archive">
  <p:input port="source" primary="true" content-types="any"
sequence="true"/>
  <p:input port="manifest" content-types="xml" sequence="true">
    <p:empty/>
  </p:input>
  <p:input port="archive" content-types="any" sequence="true">
    <p:empty/>
  </p:input>
  <p:output port="result" primary="true" content-types="any"
sequence="false"/>
  <p:output port="report" content-types="application/xml"
sequence="false"/>
  <p:option name="format" as="xs:QName" select="'zip'"/>
  <p:option name="relative-to" as="xs:anyURI?"/>
  <p:option name="parameters" as="map(xs:QName, item()*)?"/>
</p:declare-step>
```

The `p:archive` step can perform several different operations on archives. The most common one will likely be creating an archive, but it could also, depending on the archive format, provide services like update, freshen or even merge. The only format implementations **must** support is [ZIP]. The list of formats supported by the `p:archive` step is *implementation-defined*.

The `p:archive` step has the following input ports:

**`source`**
: The (primary) `source` port is used to provide documents to be archived (for instance constructed by other steps). How and which of these documents are processed is governed by the document(s) appearing on the other input ports and the combination of options and parameters. See below for details. It is a *dynamic error* (`err:XC0084`) if two or more documents appear on the `p:archive` step's `source` port that have the same base URI or if any document that appears on the `source` port has no base URI.

**`manifest`**
: The `manifest` port can receive a manifest document that tells the step how to construct the archive. If no manifest document is provided on this port, a default manifest is constructed automatically. See Section 2.3.1, “The archive manifest”. It is a *dynamic error* (`err:XC0100`) if the document on port `manifest` does not conform to the given schema.

It is a *dynamic error* (`err:XC0112`) if more than one document appears on the port `manifest`.

The default input for this port is the empty sequence.

**`archive`**

The `archive` port is used to provide the step with existing archive(s) for operations like update, freshen or merge. Handling of ZIP files supports modifying archives appearing on the `archive` port (Section 2.3.2, "Handling of ZIP archives"). The list of archive formats that can be modified by `p:archive` is *implementation-defined*. For instance an implementation that supports archive merging may accept more than one document on the `archive` port.

The default input for this port is the empty sequence.

The `p:archive` step has the following output ports:

**`result`**

The (primary) `result` port will output the resulting archive.

**`report`**

The `report` port will output a report about the archiving operation. This will be the same as the manifest (as provided on the `manifest` port or automatically created if there was no manifest provided), optionally amended with additional attributes and/or elements. The semantics of any additional attributes, elements and their values are *implementation-defined*.

The `p:archive` step has the following options:

**`format`**

The format of the archive can be specified using the `format` option. Implementations **must** support the [ZIP] format, specified with the value `zip`. It is *implementation-defined* what other formats are supported.

**`parameters`**

The `parameters` option can be used to supply parameters to control the archiving. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

**relative-to**

The relative-to option is used in creating a manifest when no manifest is provided on the manifest port. If a manifest is present this option is not used. If the option's value is a relative URI, it is made absolute against the base URI of the element on which it is specified (p:with-option or the step in case of a syntactic shortcut value). It is a *dynamic error* (err:XD0064) if the base URI is not both absolute and valid according to [RFC 3986].

The format of the archive is determined as follows:

- If the format option is specified, this determines the format of the archive. Implementations **must** support the [ZIP] format, specified with the value zip. It is *implementation-defined* what other formats are supported. It is a *dynamic error* (err:XC0081) if the format of the archive does not match the format as specified in the format option.

- If no format option is specified or if its value is the empty sequence, the archive's format will be determined by the step, using the content-type document-property of the document on the archive port and/or by inspecting its contents. It is *implementation-defined* how the step determines the archive's format. Implementations **should** recognize archives in [ZIP] format.

It is a *dynamic error* (err:XC0085) if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.

### 2.3.1. The archive manifest

An archive manifest specifies which documents will be considered in processing the archive. Every entry in the archive must have a corresponding entry in the manifest; if no such entry is provided, one will be constructed automatically (see below). If manifest entries are provided for documents that *are not* in the archive, how those are processed depends on the archive type and the parameters passed to the step.

A manifest is represented by a c:archive root element:

```
<c:archive>
    (c:entry* &
     anyNonXProcElement*)
</c:archive>
```

The `c:archive` root element may contain additional *implementation-defined* attributes.

All entries in the archive must be present as `c:entry` child elements:

```
<c:entry
  name = string
  href = anyURI
  comment? = string
  method? = string
  level? = string
  content-type? = ContentType>
    anyElement*
</c:entry>
```

- The `name` attribute specifies the name of the entry in the archive.
- The `href` attribute must be a valid URI according to [RFC 3986]. If its value is relative, it is made absolute against the base URI of the manifest. There are two possible cases:
  - If the (absolute) `href` value is exactly the same as the base URI of a document appearing on the `source` port, that document is associated with this entry. If this entry is to be added to the archive, the associated document will be used. (The `serialization` document property can be used to provide serialization properties.)
  - If no document on the `source` port has a base URI that is exactly the same as the (absolute) `href` value, the document at the specified URI is associated with this entry. These documents are stored in the archive "as is"; they **must not** be parsed and re-serialized.
- The `method` attribute specifies how the entry should be compressed. The default compression method is *implementation-defined*. Implementations **must** support no compression, specified with the value `none`. It is *implementation-defined* what other compression methods are supported.
- The `level` attribute specifies the level of compression. The default compression method is *implementation-defined*. It is *implementation-defined* what compression levels are supported.

- The `content-type` attribute specifies the content-type of the entry as detected by the processor. It will be set by `p:archive-manifest` in constructing the manifest. It will be ignored by `p:archive`.

The `p:archive` step **should** strive to retain the order of the `c:entry` elements when constructing the archive. For instance, an e-book in EPUB format has a non-compressed entry that must be first in the archive. It should be possible to construct such an archive using `p:archive`.

The `c:entry` elements may contain additional *implementation-defined* attributes.

If no manifest entry is provided for a document appearing on the `source` port, the step will create a manifest entry for the document. (If no document arrives on the `manifest` port at all, a complete manifest document will be created.)

In a constructed manifest entry:

- The entry's `href` value is the base URI of the document.
- The entry's `name` value is derived from the base URI of the document and the `relative-to` option.
    - First, the value of the `relative-to` option is made absolute. If the initial substring of the base URI is exactly the same as the resulting absolute value, then the `name` is the portion of the base URI that follows that initial substring.
    - If there is no `relative-to` option or if its value is not the initial substring of the base URI of the document, the name is the *path* portion of the URI (per [RFC 3986]). If the path portion begins with an initial slash, that slash is removed.

It is a *dynamic error* (`err:XC0118`) if an archive manifest is invalid according to the specification.

### 2.3.2. Handling of ZIP archives

The format of the archive can be specified using the `format` option. Implementations **must** support the [ZIP] format, specified with the value `zip`.

When ZIP archives are processed, every `name` in the manifest must be a relative path without a leading slash.

The `parameters` option can be used to supply parameters to control the archiving. For the `zip` format, the following parameters **must** be supported:

**`command`**
: Specifies what operation to perform. If not specified, its default value is `update`. Implementations must support the values `update`, `create`, `freshen`, and `delete`. The `p:archive` step may support additional, *implementation-defined* commands for ZIP files. Unless otherwise specified, exactly zero or one ZIP archive can appear on the `archive` port for the commands described below. If no archive appears, a new archive will be created.

  **`update`**
  : When the `command` parameter is set to `update`, the ZIP archive will be updated:

    1. For every entry in the ZIP file:
       - If the manifest contains a `c:entry` with a matching `name`, the entry in the ZIP file is updated with the document identified by the `c:entry` in the manifest.
       - If the manifest does not contain a matching `c:entry`, the ZIP entry name is resolved against the base URI of the ZIP file.
         - If a document exists at that URI and either has no timestamp or has a timestamp more than the timestamp in the ZIP file, the entry in the ZIP file will be updated with the document at the resolved URI.
         - If no document exists at that URI, or the document cannot be accessed, or the document has a timestamp and the timestamp in the ZIP archive is more recent than the timestamp of the document, then the ZIP entry is unchanged.
    2. For every `c:entry` in the manifest that does not have a matching entry in the ZIP file, the ZIP file will be updated by adding the document identified by the `c:entry` to the ZIP file.

**`create`**
When the `command` parameter is set to `create`, the ZIP archive will be created. Creating a ZIP archive behaves exactly like `update` except that any timestamps are ignored; every ZIP entry will be updated or created if there is a `c:entry` or matching document for it. The document must be obtained by dereferencing the URI in `href`. It is a *dynamic error* (`err:XD0011`) if the resource referenced by the `href` option does not exist, cannot be accessed or is not a file.

**`freshen`**
When the `command` parameter is set to `freshen`, existing files in the ZIP archive may be updated, but no new files will be added. Freshing a ZIP archive behaves exactly like `update` except that only entries that already exist in the ZIP archive are considered.

**`delete`**
When the `command` parameter is set to `delete`, exactly one document in ZIP format must appear on the `archive` port. For every entry in the ZIP file:

- If the manifest contains a `c:entry` with a matching `name`, the entry in the ZIP file is removed from the ZIP archive.

If the manifest contains `c:entry` elements which do not have a matching entry in the ZIP archive, they are simply ignored.

**`level`**
Specifies the default compression level for files added to or updated in the archive. If the `level` attribute is specified on a `c:entry`, its value takes precedence for that entry. Values that must be supported for ZIP files are: "`smallest`", "`fastest`", "`default`", "`huffman`", and "`none`".

**`method`**
Specifies the default compression method for files added to or updated in the archive. If the `method` attribute is specified on a `c:entry`, its value takes precedence for that entry. Values that must be supported for ZIP files are: "`none`" and "`deflated`".

It is a *dynamic error* (`err:XC0080`) if the number of documents on the `archive` does not match the expected number of archive input documents for the given `format` and `command`.

Implementations of other archive formats **should** use the same parameter names if applicable. The value spaces for these parameters may be format-specific though. The actual parameter names supported by `p:archive` for a particular format are *implementation-defined*.

#### Document properties

No document properties are preserved. The archive has no `base-uri`.

## 2.4. p:archive-manifest

The `p:archive-manifest` creates an XML manifest file describing the contents of the archive appearing on its `source` port.

```
<p:declare-step type="p:archive-manifest">
  <p:input port="source" primary="true" content-types="any"
sequence="false"/>
  <p:output port="result" primary="true" content-types="application/xml"
sequence="false"/>
  <p:option name="format" as="xs:QName?"/>
  <p:option name="parameters" as="map(xs:QName, item()*)?"/>
  <p:option name="relative-to" as="xs:anyURI?"/>
  <p:option name="override-content-types" as="array(array(xs:string))?"/>
</p:declare-step>
```

The `p:archive-manifest` step inspects the archive appearing on its `source` port and outputs a manifest describing the contents of the archive on its `result` port.

The format of the archive is determined as follows:

- If the `format` option is specified, this determines the format of the archive. Implementations **must** support the [ZIP] format, specified with the value `zip`. It is *implementation-defined* what other formats are supported.
- If no `format` option is specified or if its value is the empty sequence, the archive's format will be determined by the step, using the `content-type`

document-property of the document on the `source` port and/or by inspecting its contents. It is *implementation-defined* how the step determines the archive's format. Implementations **should** recognize archives in [ZIP] format.

It is a *dynamic error* (`err:XC0085`) if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.

The `parameters` option can be used to supply parameters to control the archive manifest generation. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

The `relative-to` option, when present, is used in creating the value of the manifest's `c:entry/@href` attribute. If the option is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or the step in case of a syntactic shortcut value). It is a *dynamic error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

The generated manifest has the format as described in Section 2.3.1, "The archive manifest". Implementations **must** supply an `c:entry` element and its `name` and `content-type` attributes for every entry in the archive. The value of the generated manifest's `c:entry/@href` attribute will be determined in the same way as a base URI of an unarchived document by Section 2.38, "p:unarchive". It is a *dynamic error* (`err:XC0120`) if the `relative-to` option is not present and the document on the `source` port does not have a base URI. Additional information provided for entries in `p:archive-manifest` is *implementation-defined*.

#### 2.4.1. Overriding content types

The `override-content-types` option can be used to partially override the content-type determination mechanism. If present, it must be an array of arrays, where the inner arrays consist of exactly two strings:

- The first member in an inner array **must** be a regular expression as specified in [XPath and XQuery Functions and Operators 3.1], section 7.61 "`Regular`

`Expression Syntax`". It is a *dynamic error* (`err:XC0147`) if the specified value is not a valid XPath regular expression.

- The second member in an inner array **must** be a valid a MIME content-type. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*`type/subtype+ext`*" or "*`type/subtype`*".

It is a *dynamic error* (`err:XC0146`) if the specified value for the `override-content-types` option is not an array of arrays, where the inner arrays have exactly two members of type `xs:string`.

Determining an archive entry's content-type is as follows:

- The XPath regular expressions (the first members of the inner arrays) will be matched against the path of the entry *in* the archive. This will be done in the order of appearance in the outer array (so order is significant). The matching is done unanchored: it is a match if the regular expression matches part of the entry's path. Informally: matching behaves like applying the XPath `matches#2` function, like in `matches($path-in-archive, $regular-expression)`.

> **Note**
>
> Depending on how archives are constructed, the path of an entry in an archive can be with or without a leading slash. Usually it will be without. For archives constructed by `p:archive` no leading slash will be present.

- If a match is found, the content-type (the second member of the inner array for which the match was found) is used as the entry's content-type.
- If no match was found for all inner arrays, the normal (implementation-defined) mechanism for determining the content-type is used.

For example: setting the `override-content-types` option to `[ ['.rels$', 'application/xml'], ['^special/', 'application/octet-stream'] ]` means that all files ending with `.rels` will get the content-type `application/xml`. All files

in the archive's `special` directory (including sub-directories) will get the content-type `application/octet-stream`.

#### Document properties

No document properties are preserved. The manifest has no `base-uri`.

## 2.5. p:cast-content-type

The `p:cast-content-type` step creates a new document by changing the media type of its input. If the value of the `content-type` option and the current media type of the document on `source` port are the same, this document will appear unchanged on `result` port.

```
<p:declare-step type="p:cast-content-type">
  <p:input port="source" content-types="any"/>
  <p:output port="result" content-types="any"/>
  <p:option name="content-type" required="true" as="xs:string"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
</p:declare-step>
```

The input document is transformed from one media type to another. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*type/subtype+ext*" or "*type/subtype*". It is a *dynamic error* (`err:XC0071`) if the `p:cast-content-type` step cannot perform the requested cast.

The `parameters` can be used to supply parameters to control casting. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

### 2.5.1. Casting from an XML media type

- Casting from one XML media type to another simply changes the "`content-type`" document property.

- Casting from an XML media type to an HTML media type changes the "`content-type`" document property and removes any serialization property.

- Casting from an XML media type to a JSON media type converts the XML into JSON. The precise nature of the conversion from XML to JSON is *implementation-defined*. If the input document is an XML representation of JSON as defined in [XPath and XQuery Functions and Operators 3.1], implementations **must** produce the same result as `fn:parse-json(fn:xml-to-json())` by default. If the input document has a `c:param-set` document element, an instance of `map(xs:QName, xs:string)` **must** be returned that represents the document's `c:param` elements. The serialization property is removed.

- Casting from an XML media type to a text media type serializes the XML document by calling `fn:serialize($doc, $param)` where `$doc` is the document on the `source` port and `$param` is the serialization property of this document. The resulting string is wrapped by a document node and returned on the `result` port. The serialization property is removed.

- Casting from an XML media type to any other media type **must** support the case where the input document is a `c:data` document. The resulting document will have the specified media type and a representation that is the content of the `c:data` element after decoding the base64 encoded content The serialization property is removed.

  It is a *dynamic error* (`err:XC0072`) if the `c:data` contains content is not a valid base64 string.

  It is a *dynamic error* (`err:XC0073`) if the `c:data` element does not have a `content-type` attribute.

  It is a *dynamic error* (`err:XC0074`) if the `content-type` is supplied and is not the same as the `content-type` specified on the `c:data` element.

  Casting from an XML media type to any other media type when the input document is not a `c:data` document is *implementation-defined*.

### 2.5.2. Casting from an HTML media type

- Casting from an HTML media type to an XML media type changes "`content-type`" document property and removes any serialization property.
- Casting from an HTML media type to another HTML media type changes "`content-type`" document property.
- Casting from an HTML media type to a JSON media type is *implementation-defined*.
- Casting an an HTML media type to a text media type serializes the HTML document by calling `fn:serialize($doc, $param)` where `$doc` is the document on the `source` port and `$param` is the serialization property of this document. The resulting string is wrapped by a document node and returned on the `result` port. The serialization property is removed.
- Casting from an HTML media type to any other media type is *implementation-defined*.

### 2.5.3. Casting from a JSON media type

- Casting from a JSON media type to an XML media type converts the JSON into XML. An implementation **must** support the format specified in section "XML Representation of JSON" of [XPath and XQuery Functions and Operators 3.1] as default for the resulting XML. It is *implementation-defined* whether other result formats are supported. The serialization property is removed.
- Casting from a JSON media type to an HTML media type is *implementation-defined*.
- Casting from a JSON media type to another JSON media type changes "`content-type`" document property.
- Casting from a JSON media type to a text media type serializes the JSON document by calling `fn:serialize($doc, $param)` where `$doc` is the document on the `source` port and `$param` is the serialization property of this document. The resulting string is wrapped by a document node and returned on the `result` port. The serialization property is removed.

- Casting from a JSON media type to any other media type is *implementation-defined*.

### 2.5.4. Casting from a text media type

- Casting from a text media type to an XML media type parses the text value of the document on `source` port by calling `fn:parse-xml`. It is a *dynamic error* (`err:XD0049`) if the text value is not a well-formed XML document. The serialization property is removed.

- Casting from a text media type to an HTML media type parses the text value of the document on `source` port into an XPath data model document that contains a tree of elements, attributes, and other nodes. The precise way in which text documents are parsed into the XPath data model is *implementation-defined*. It is a *dynamic error* (`err:XD0060`) if the text document can not be converted into the XPath data model. The serialization property is removed.

- Casting from a text media type to a JSON media type parses the text value of the document on `source` port by calling `fn:parse-json($doc, $par)` where `$doc` is the text document and `$par` is the `parameter` option. It is a *dynamic error* (`err:XD0057`) if the text document does not conform to the JSON grammar, unless the parameter liberal is true and the processor chooses to accept the deviation. It is a *dynamic error* (`err:XD0058`) if the parameter duplicates is reject and the text document contains a JSON object with duplicate keys. It is a *dynamic error* (`err:XD0059`) if the parameter map contains an entry whose key is defined in the specification of `fn:parse-json` and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter `escape` with `true()` is also present. The serialization property is removed.

- Casting from a text media type to another text media type changes "`content-type`" document property.

- Casting from a text media type to any other media type is *implementation-defined*.

#### 2.5.5. Casting from any other media type

- Casting from a non-XML media type to an XML media type produces an XML document with a `c:data` document element. The original media type will be preserved in the `content-type` attribute on the `c:data` element.

```
<c:data
  content-type = ContentType
  charset? = string
  encoding? = string>
    string
</c:data>
```

  The content of the `c:data` element is the base64 encoded representation of the non-XML content. The serialization property is removed.

- Casting from any other media type to a HTML media type, a JSON media type or a text document is *implementation-defined*.

- Casting from any other media type to any other media type is *implementation-defined*.

#### Document properties

All document properties are preserved except the `content-type` property which is updated accordingly and the `serialization` property which is removed by some casting methods.

## 2.6. p:compare

The `p:compare` step compares two documents for equality.

```
<p:declare-step type="p:compare">
  <p:input port="source" primary="true" content-types="any"/>
  <p:input port="alternate" content-types="any"/>
  <p:output port="result" content-types="application/xml"/>
  <p:output port="differences" content-types="any" sequence="true"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="method" as="xs:QName?"/>
  <p:option name="fail-if-not-equal" as="xs:boolean" select="false()"/>
</p:declare-step>
```

This step takes single documents on each of two ports and compares them. If `method` is not specified, or if `deep-equal` is specified, the comparison uses `fn:deep-equal` (as defined in [XPath and XQuery Functions and Operators 3.1]). Implementations of `p:compare` **must** support the `deep-equal` `method`; other supported methods are *implementation-defined*. It is a *dynamic error* (`err:XC0076`) if the comparison `method` specified in `p:compare` is not supported by the implementation. It is a *dynamic error* (`err:XC0077`) if the media types of the documents supplied are incompatible with the comparison `method`.

It is a *dynamic error* (`err:XC0019`) if the documents are not equal according to the specified comparison `method`, and the value of the `fail-if-not-equal` option is `true`. If the documents are equal, or if the value of the `fail-if-not-equal` option is `false`, a `c:result` document is produced with contents `true` if the documents are equal, otherwise `false`.

If `fail-if-not-equal` is `false`, and the documents differ, an *implementation-defined* summary of the differences between the two documents may appear on the `differences` port.

#### Document properties

No document properties are preserved. The comparison document has no `base-uri`.

## 2.7. p:compress

The `p:compress` step serializes the document appearing on its `source` port and outputs a compressed version of this on its `result` port.

```
<p:declare-step type="p:compress">
  <p:input port="source" primary="true" content-types="any"
sequence="false"/>
  <p:output port="result" primary="true" content-types="any"
sequence="false"/>
  <p:option name="format" as="xs:QName" select="'gzip'"/>
  <p:option name="serialization" as="map(xs:QName,item()*)?"/>
  <p:option name="parameters" as="map(xs:QName, item()*)?"/>
</p:declare-step>
```

The `p:compress` step first serializes the document appearing on its `source`. It then compresses the outcome of this serialization and outputs the result on its `result` port.

The `p:compress` step has the following options:

**`format`**
: The format of the compression can be specified using the `format` option. Implementations **must** support the [GZIP] format, specified with the value `gzip`. It is *implementation-defined* what other formats are supported. It is a *dynamic error* (`err:XC0202`) if the compression format cannot be understood, determined and/or processed.

**`parameters`**
: The `parameters` option can be used to supply parameters to control the compression. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

**`serialization`**
: The `serialization` option is provided to control the serialization of content before compression takes place. If the document to be stored has a `serialization` property, the serialization is controlled by the merger of the two maps where the entries in the `serialization` property take precedence. Serialization is described in [XProc 3.0].

#### Document properties

All document properties are preserved, except for the `content-type` property which is updated accordingly and the `serialization` property which is removed.

## 2.8. p:count

The `p:count` step counts the number of documents in the `source` input sequence and returns a single document on `result` containing that number. The generated

document contains a single `c:result` element whose contents is the string representation of the number of documents in the sequence.

```
<p:declare-step type="p:count">
  <p:input port="source" content-types="any" sequence="true"/>
  <p:output port="result" content-types="application/xml"/>
  <p:option name="limit" as="xs:integer" select="0"/>
</p:declare-step>
```

If the `limit` option is specified and is greater than zero, the `p:count` step will count at most that many documents. This provides a convenient mechanism to discover, for example, if a sequence consists of more than 1 document, without requiring every single document to be buffered before processing can continue.

#### Document properties

No document properties are preserved. The count document has no `base-uri`.

## 2.9. p:delete

The `p:delete` step deletes items specified by a *selection pattern* from the `source` input document and produces the resulting document, with the deleted items removed, on the `result` port.

```
<p:declare-step type="p:delete">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="match" required="true" as="xs:string"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. A *selection pattern* may match multiple items to be deleted.

If an element is selected by the `match` option, the entire subtree rooted at that element is deleted.

It is a *dynamic error* (`err:XC0023`) if the `match` option matches the document node.

This step cannot be used to remove namespaces. It is a *dynamic error* (`err:XC0062`) if the `match` option matches a namespace node. Also, note that deleting an attribute named `xml:base` does not change the base URI of the element on which it occurred.

**Document properties**

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. In all other cases, all document properties are preserved.

## 2.10. p:error

The `p:error` step generates a *dynamic error* using the input provided to the step.

```
<p:declare-step type="p:error">
  <p:input port="source" sequence="true" content-types="text xml"/>
  <p:output port="result" sequence="true" content-types="any"/>
  <p:option name="code" required="true" as="xs:QName"/>
</p:declare-step>
```

This step uses the document provided on its input as the content of the error raised. An instance of the `c:errors` element will be produced on the error output port, as is always the case for *dynamic errors*. The error generated can be caught by a `p:try` just like any other dynamic error.

For authoring convenience, the `p:error` step is declared with a single, primary output port. With respect to connections, this port behaves like any other output port even though nothing can ever appear on it since the step always fails.

For example, given the following invocation:

```
<p:error xmlns:my="http://www.example.org/error"
         name="bad-document" code="my:unk12">
   <p:with-input port="source">
     <message>The document element is unknown.</message>
   </p:with-input>
</p:error>
```

The error vocabulary element (and document) generated on the error output port would be:

```
<c:errors xmlns:c="http://www.w3.org/ns/xproc-step"
          xmlns:p="http://www.w3.org/ns/xproc"
          xmlns:my="http://www.example.org/error">
 <c:error name="bad-document" type="p:error"
          code="my:unk12"><message
          >The document element is unknown.</message>
</c:error>
</c:errors>
```

The `href`, `line` and `column`, or `offset`, might also be present on the `c:error` to identify the location of the `p:error` element in the pipeline.

#### Document properties

No document properties are preserved but that's irrelevant as no document is ever produced.

### 2.11. p:filter

The `p:filter` step selects portions of the source document based on a (possibly dynamically constructed) XPath select expression.

```
<p:declare-step type="p:filter">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" sequence="true" content-types="text xml html"/>
  <p:option name="select" required="true" as="xs:string"/>
</p:declare-step>
```

This step behaves just like an `p:input` with a `select` expression except that the select expression is computed dynamically.

#### Document properties

No document properties are preserved. The `base-uri` property of each document will reflect the base URI of the selected node(s).

## 2.12. p:hash

The `p:hash` step generates a hash, or digital "fingerprint", for some value and injects it into the `source` document.

```
<p:declare-step type="p:hash">
  <p:input port="source" primary="true" content-types="xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="value" required="true" as="xs:string"/>
  <p:option name="algorithm" required="true" as="xs:QName"/>
  <p:option name="match" as="xs:string" select="'/*/node()'"/>
  <p:option name="version" as="xs:string?"/>
</p:declare-step>
```

The value of the `algorithm` option must be a QName. If it does not have a prefix, then it must be one of the following values: "crc", "md", or "sha".

If a `version` is not specified, the default version is algorithm-defined. For "`crc`" it is 32, for "`md`" it is 5, for "`sha`" it is 1.

A hash is constructed from the string specified in the `value` option using the specified algorithm and version. Implementations **must** support [CRC32], [RFC 1321], and [SHA1] hashes. It is *implementation-defined* what other algorithms are supported. The resulting hash **should** be returned as a string of hexadecimal characters.

The value of the `match` option must be an XSLTSelectionPattern.

The hash of the specified value is computed using the algorithm and parameters specified. It is a *dynamic error* (`err:XC0036`) if the requested hash algorithm is not one that the processor understands or if the value or parameters are not appropriate for that algorithm.

The matched nodes are specified with the *selection pattern* in the `match` option. For each matching node, the string value of the computed hash is used in the output (if more than one node matches, the *same* hash value is used in each match). Nodes that do not match are copied without change.

If the expression given in the `match` option matches an *attribute*, the hash is used as the new value of the attribute in the output. If the attribute is named "`xml:base`", the base URI of the element **must** also be amended accordingly.

If the document node is matched, the entire document is replaced by a text node with the hash. What appears on port `result` is a text document with the text node wrapped in a document node.

If the expression matches any other kind of node, the entire node (and *not* just its contents) is replaced by the hash.

#### Document properties

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. For other document types, all document properties are preserved.

## 2.13. p:http-request

The `p:http-request` step allows authors to interact with resources over HTTP or related protocols. Implementations **must** support the `http` and `https` protocols. (Implementors are encouraged to support as many protocols as practical. In particular, pipeline authors may attempt to use `p:http-request` to load documents with computed URIs using the `file:` scheme.)

```
<p:declare-step type="p:http-request">
  <p:input port="source" content-types="any" sequence="true"/>
  <p:output port="result" primary="true" content-types="any"
sequence="true"/>
  <p:output port="report" content-types="application/json"/>
  <p:option name="href" as="xs:anyURI" required="true"/>
  <p:option name="method" as="xs:string?" select="'GET'"/>
  <p:option name="serialization" as="map(xs:QName,item()*)?"/>
  <p:option name="headers" as="map(xs:string, xs:string)?"/>
  <p:option name="auth" as="map(xs:string, item()+)?"/>
  <p:option name="parameters" as="map(xs:QName, item()*)?"/>
  <p:option name="assert" as="xs:string" select="'.?status-code lt 400'"/>
</p:declare-step>
```

The `p:http-request` step performs the HTTP request specified by the `method` option against the URI specified in the `href` option. In simple cases, for example, a GET request on an unauthenticated URI, nothing else is necessary to form a complete request.

If the method, for example, POST, supports a body, the request body is constructed using the document(s) appearing on the `source` port. For the convenience of pipeline authors, documents may appear on the `source` port even when the request method (such as GET or HEAD) does not define the semantics of a payload. If the semantics are undefined, the documents are ignored when constructing the request unless the `parameters` option specifies "`send-body-anyway`" as `true()`.

The headers for the request come from the `headers` option (see below). If exactly one document appears on the `source` port, its document properties also contribute to the overall request headers.

The response from the HTTP request appears on the `result` and `report` ports. Any documents contained in the response body will appear on the `result` port. Each document in the response will be parsed according to its content-type (but see "`override-content-type`" in the `parameters` option). Details about the outcome of the request will appear as a map on the `report` port. The map will always contain:

**`status-code` (an `xs:integer`)**
: This is the HTTP status code returned for the request.

**`base-uri` (an `xs:anyURI`)**
: This is the URI of the last request made and is always available in the report even when the request does not return any documents. In the case of HTTP redirection, the base URI returned may be different from the original request URI.

**`headers` (a `map(xs:string, xs:string)`)**
: These are the HTTP headers returned for the request. The map may be empty. Header names are converted to lowercase.

The `p:http-request` step has the following options:

**`href`**
: The `href` option specifies the request's IRI. Relative values are resolved against the base URI of the element on which the option is specified (the relevant `p:with-option` or the step element in the case of a syntactic shortcut value).

  Fragment identifiers are removed before making the request. Query parameters are passed through unchanged. It is a *dynamic error* (`err:XC0128`) if the URI's scheme is unknown or not supported. It is the pipeline author's responsibility to

escape problematic UTF-8 characters in the `href` value, for example with `escape-html-uri()`.

**`method`**

The `method` specifies the HTTP request method. The value is implicitly turned into an uppercase string if necessary. It is *implementation defined* which HTTP methods are supported. An implementation **should** implement at least the methods `GET`, `POST`, `PUT`, `DELETE`, and `HEAD` (for HTTP and HTTPS). It is a *dynamic error* (`err:XC0122`) if the given method is not supported.

**`serialization`**

The `serialization` option is used to control the serialization of documents for the request body. If a document has a "`serialization`" document property, the effective value of the serialization options is the union of the two maps, where the entries in the "`serialization`" document property take precedence.

**`headers`**

The key/value pairs in the `headers` map are used to construct the request headers. Each map key is used as a header name and the value associated with that key in the map is used as the header value.

If a single document appears on the `source` port, then document properties on that document may be added as additional headers. For XML, HTML, and text documents with a `serialization` document property having an `encoding` key, a `charset` is appended to the created `content-type` header of the HTTP request. Properties in the `http://www.w3.org/ns/xproc-http` namespace will be added to the headers, using the local-name of the property QName as the header name. These properties are only copied if they are not specified in the `header` map. In other words, if the same header name appears in both places, the value from the map is used and the value from the document properties is ignored. (Header names are case-insensitive, so a case-insensitive comparison must be performed.) If multiple documents appear on the `source` port, none of their properties are used in the request headers.

The behavior of the `p:http-request` depends on the headers specified. In particular:

**content-type**
: If a content-type header is provided, it will be used. For a single document request, this overrides the content type value of the document. If the content type specified begins with "multipart/", a multipart request will be sent to the server.

  It is a *dynamic error* (err:XD0079) if a supplied content-type is not a valid media type of the form "*type/subtype+ext*" or "*type/subtype*".

**transfer-encoding**
: If a transfer-encoding header is provided, the request **must** be sent with that encoding. It is a *dynamic error* (err:XC0131) if the processor cannot support the requested encoding.

**authorization**
: The authorization header is used to authenticate a request. If the auth *option* is specified, any key or property that would have contributed a header named "authorization" (irrespective of case) is ignored. The authorization header is determined exclusively by the auth option when it is present.

HTTP headers are case-insensitive but keys in maps are not; be careful when specifying the request headers. It is a *dynamic error* (err:XC0127) if the headers map contains two keys that are the same when compared in a case-insensitive manner. (That is, when fn:uppercase($key1) = fn:uppercase($key2).)

**auth**
: Many web services are only available to authenticated users, that is, to users who have "logged in". The auth option allows the pipeline author to specify information that may be required to generate an "Authorization" header. The standard values support HTTP "Basic" and "Digest" authentication, but other authentication methods are allowed.

  The following standard keys are defined:

  **username (xs:string)**
  : The username.

  **password (xs:string)**
  : The password associated with the username.

**`auth-method` (`xs:string`)**
: The authentication method. Appropriate values for the "`auth-method`" key are "`Basic`" or "`Digest`" but other values are allowed. If the authentication method is "`Basic`" or "`Digest`", authentication is handled as per [RFC 2617]. The interpretation of values associated with the "`auth-method`" key other than "`Basic`" or "`Digest`" is *implementation defined*.

**`send-authorization` (`xs:boolean`)**
: The "`send-authorization`" key can be used to attempt to allow the request to avoid an authentication challenge. If the "`send-authorization`" key is "`true()`", and the authentication method specified by the value associated with the "`auth-method`" key supports generation of an "`Authorization`" header without a challenge, then the header is generated and sent on the first request. If the "`send-authorization`" key is absent or does not have the value "`true`", the first request is sent without an "`Authorization`" header.

Other key value pairs in map "`auth`" are *implementation defined*. It is a *dynamic error* (`err:XC0123`) if any key in the "`auth`" map is associated with a value that is not an instance of the required type.

If the initial response to the request is an authentication challenge, the values provided in the `auth` map and any relevant data from the challenge are used to generate an "`Authorization`" header and the request is sent again. If that authorization fails, the request is not retried.

It is a *dynamic error* (`err:XC0003`) if a "`username`" or a "`password`" key is present without specifying a value for the "`auth-method`" key, if the requested `auth-method` isn't supported, or the authentication challenge contains an authentication method that isn't supported. All implementations **must** support "Basic" and "Digest" authentication per [RFC 2617].

**parameters**
: The `parameter` option can be used to provide values for fine tuning the construction of the request and/or handling of the server response. A number of parameters are defined in this specification. It is *implementation defined* which other key/value pairs in the `parameters` option are supported.

**override-content-type (`xs:string`)**
Ordinarily, the value of the `content-type` header provided in the server response controls the interpretation of any body in the response. If the "`override-content-type`" parameter is provided, then its value is used to interpret the body. The content-type header that appears on the `report` port is not changed. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*`type/subtype+ext`*" or "*`type/subtype`*". It is a *dynamic error* (`err:XC0030`) if the response body cannot be interpreted as requested (e.g. `application/json` to override `application/xml` content).

**http-version (`xs:string`)**
The `http-version` parameter indicates which version of HTTP **must** be used for the request.

**accept-multipart (`xs:boolean`)**
If the `accept-multipart` parameter is present and explicitly has the value `false()`, a dynamic error will be raised, if a multipart response is received from the server. This feature is a convenience for pipeline authors as it will raise an error when the multipart request is received, rather than having the presence of a sequence raise an error further along in the pipeline, or simply producing anomalous results. It is a *dynamic error* (`err:XC0125`) if the key "`accept-multipart`" as the value `false()` and a multipart response is detected.

**override-content-encoding (`xs:string`)**
If the "`override-content-encoding`" parameter is present, the response will be treated as if the response contained a "`content-encoding`" header with the specified value. The content-encoding header that appears on the `report` port is not changed. It is a *dynamic error* (`err:XC0132`) if the override content encoding cannot be supported.

**permit-expired-ssl-certificate (`xs:boolean`)**
If "`permit-expired-ssl-certificate`" is true, then the processor should not reject responses where the server provides an expired SSL certificate.

**permit-untrusted-ssl-certificate (`xs:boolean`)**
If "`permit-untrusted-ssl-certificate`" is true, then the processor should not reject response where the server provides an SSL certificate

which is not trusted, for example, because the certificate authority (CA) is unknown.

**follow-redirect (`xs:integer`)**
The "`follow-redirect`" parameter allows the pipeline author to specify the step's behaviour in the case of a redirect response. A value of `0` indicates that redirects are not to be followed, `-1` indicates that redirects are to be followed indefinitely, and a specific number indicates the maximum number of redirects to follow. The default behaviour in case of a redirect response is *implementation defined*.

**timeout (`xs:integer`)**
If a "`timeout`" is specified, it **must** be a non-negative integer. It controls the time the XProc processor waits for the request to be answered. If a value is given, it is taken as the number of seconds to wait for the response to be delivered. If no response is received after that time, the request is terminated and a status-code `408` is assumed.

**fail-on-timeout (`xs:boolean`)**
If "`fail-on-timeout`" is true, a dynamic error is raised if a `408` response is received (either as a consequence of setting a value for the "`timeout`" parameter or as status code returned by a server). It is a *dynamic error* (`err:XC0078`) if the value associated with the "`fail-on-timeout`" is associated with `true()` and a HTTP status code `408` is encountered. If "`fail-on-timeout`" is true, it prevents any dynamic error with code `C0126` resulting from the `assert` option to be raised for request's timeout.

> **Note**
>
> Please note that the "`fail-on-timeout`" parameter is different from the "`timeout`" option on the `p:http-request` step (see *Controlling long running steps* in *XProc 3.0: An XML Pipeline Language*). If the *step* does not finish in the specified time, `D0053` is raised. If the *request* does not finish in time, and `fail-on-timeout` is true, `C0078` is raised. The actual times after which a timeout is detected may also differ slightly.

**status-only (`xs:boolean`)**
: If the "`status-only`" parameter is true, this indicates that the pipeline author is only interested in the response code. An empty sequence is always returned on the `result` port in this case. The implementation may save resources by ignoring the response body. The map on the `report` will contain the status code and an empty map for "`headers`".

**suppress-cookies (`xs:boolean`)**
: If the "`suppress-cookies`" parameter is true, the implementation **must not** send any cookies with the request.

**send-body-anyway (`xs:boolean`)**
: If the "`send-body-anyway`" parameter is true, and one or more documents appear on the `source` port, a request body is constructed from the documents and sent with the request, even if the semantics of sending a body are not specified for the HTTP method in use.

It is a *dynamic error* (`err:XC0124`) if any key in the "parameters" map is associated with a value that is not an instance of the required type.

**assert (`xs:string`)**
: The `assert` option can be used by pipeline authors to raise a dynamic error if the response does not fulfill the expectations of the receiver. The option's value (if present) is interpreted as an XPath expression which will be executed using the map that appears on the `report` port as its context item. If the effective boolean value of the expression is `false()`, a dynamic error is raised. It is a *dynamic error* (`err:XC0126`) if the XPath expression in `assert` evaluates to `false`. Implementations **should** provide an XML representation of the map used as the context item with the error document to enable pipelines to access the error's cause.

### 2.13.1. Construction of a multipart request

If more than one document appears on the `source` port, or if the specified "`content-type`" header begins "`multipart/`", a multipart request will be constructed, per [RFC 1521]. The content type of the request is derived from the "`content-type`" header:

- If the "`content-type`" header specifies a multipart content type, that value will be used as the content type. If the header includes a `boundary` parameter, that value will be used as the boundary. It is a *dynamic error* (`err:XC0203`) if the specified boundary is not valid (for example, if it begins with two hyphens "--").
- If the "`content-type`" header is not specified, "`multipart/mixed`" will be used.
- It is a *dynamic error* (`err:XC0133`) if more than one document appears on the `source` port and a `content-type` header is present and the content type specified is not a multipart content type.

A multipart request must have a boundary marker, if one isn't specified in the content type, the implementation **must** construct one. It is *implementation-defined* how a multipart boundary is constructed. Implementations *are not* required to guarantee that the constructed value does not appear accidentally in the multipart data. If it does, the request will be malformed; pipeline authors must provide a boundary if they wish to assure that this cannot happen.

Each document in the sequence is serialized. If the document has a "`serialization`" document property, its values are used to determine how serialization is performed.

All of the document properties in the `http://www.w3.org/ns/xproc-http` namespace will be added as headers for the part, using the local-name of the property QName as the header name. In particular, this is how the "`id`", "`description`", "`disposition`" and other multipart headers can be provided.

### 2.13.2. Managing a multipart response

When a multipart response is received, each part is interpreted according to it's content type and a pipeline document is constructed. Any additional headers associated with the part are added to the document properties of the constructed document.

The multipart response is the resulting sequence of documents.

#### Document properties

No document properties are preserved.

## 2.14. p:identity

The `p:identity` step makes a verbatim copy of its input available on its output.

```
<p:declare-step type="p:identity">
  <p:input port="source" sequence="true" content-types="any"/>
  <p:output port="result" sequence="true" content-types="any"/>
</p:declare-step>
```

If the implementation supports passing PSVI annotations between steps, the `p:identity` step **must** preserve any annotations that appear in the input.

#### Document properties

All document properties are preserved.

## 2.15. p:insert

The `p:insert` step inserts the `insertion` port's document into the `source` port's document relative to the matching elements in the `source` port's document.

```
<p:declare-step type="p:insert">
  <p:input port="source" primary="true" content-types="xml html"/>
  <p:input port="insertion" sequence="true" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
  <p:option name="position" values="('first-child','last-
child','before','after')" select="'after'"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if that pattern matches an attribute or a namespace node. Multiple matches are allowed, in which case multiple copies of the `insertion` documents will occur. If no elements match, then the document is unchanged.

The value of the `position` option **must** be an NMTOKEN in the following list:

- “first-child” - the insertion is made as the first child of the match;
- “last-child” - the insertion is made as the last child of the match;
- “before” - the insertion is made as the immediate preceding sibling of the match;
- “after” - the insertion is made as the immediate following sibling of the match.

It is a *dynamic error* (err:XC0025) if the *selection pattern* matches anything other than an element or a document node and the value of the position option is “first-child” or “last-child”. It is a *dynamic error* (err:XC0024) if the *selection pattern* matches a document node and the value of the position is “before” or “after”.

As the inserted elements are part of the output of the step they are not considered in determining matching elements. If an empty sequence appears on the insertion port, the result will be the same as the source.

#### Document properties

All document properties on the source port are preserved. The document properties on the insertion port are not preserved or present in the result document.

## 2.16. p:json-join

The p:json-join step joins the sequence of documents on port source into a single JSON document (an array) appearing on port result. If the sequence on port source is empty, the empty sequence is returned on port result.

```
<p:declare-step type="p:json-join">
  <p:input port="source" sequence="true" content-types="any"/>
  <p:output port="result" content-types="application/json"/>
  <p:option name="flatten-to-depth" as="xs:string?" select="'0'"/>
</p:declare-step>
```

The step inspects the documents on port source in turn to create the resulting array:

- If the document under inspection is a JSON document representing an array, the array is copied to the resulting array according to the setting of option `flatten-to-depth`.
- For every other type of JSON document, for XML documents, HTML documents, or text documents, their XDM representation is appended to the resulting array.
- It is *implementation defined* if `p:json-join` is able to process document types not mentioned yet, i.e. types of binary documents. If a processor supports a given type of documents, an entry is created as described above. It is a *dynamic error* (`err:XC0111`) if a document of an unsupported document type appears on port `source` of `p:json-join`.

The option `flatten-to-depth` controls whether and to which depth members of an array appearing on port `source` are flattened. It is a *dynamic error* (`err:XC0119`) if `flatten` is neither "`unbounded`", nor a string that may be cast to a non-negative integer. An integer value of `0`, which is the default, means that no flattening takes place, so the array appearing on port `source` will be contained as an array in the resulting array. An integer value of `1` means that an array on port `source` is flattened, i.e. the members of that array will appear as individual members in the resulting array. Any value greater than `1` means that the flattening is applied recursively to arrays in arrays up to the given depth. A value of "`unbounded`" means that all arrays in arrays will be flattened. As a consequence, the resulting array appearing on port `result` will not have any arrays as members.

#### Document properties

No document properties are preserved. The joined document has no `base-uri`.

## 2.17. p:json-merge

The `p:json-merge` step merges the sequence of appearing on port `source` into a single JSON object appearing on port `result`. If the sequence on port `source` is empty, the empty sequence is returned on port `result`.

```
<p:declare-step type="p:json-merge">
  <p:input port="source" sequence="true" content-types="any"/>
  <p:output port="result" content-types="application/json"/>
  <p:option name="duplicates" values="('reject', 'use-first', 'use-last',
'use-any', 'combine')" select="'use-first'"/>
  <p:option name="key" as="xs:string" select="'concat("_",$p:index)'"/>
</p:declare-step>
```

The step inspects the documents on port `source` in turn to create the resulting map:

- If the document under inspection is a JSON document representing a map, all key-value pairs are copied into the result map unless this map already contains an entry with the given key. In this case the value of option `duplicates` determines the policy for handling duplicate keys as specified for function `map:merge` in [XPath and XQuery Functions and Operators 3.1]. It is a *dynamic error* (`err:XC0106`) if duplicate keys are encountered and option `duplicates` has value "`reject`".

- For every other type of JSON document, for XML documents, HTML documents, or text documents a new key-value pair is created and put into the resulting map. The key is created by evaluating the XPath expression in option `key` with the inspected document as context item. If the evaluation result is a single atomic value, it is taken as key. If the evaluation result is a node, its string value is taken as key. It is a *dynamic error* (`err:XC0110`) if the evaluation of the XPath expression in option `key` for a given item returns either a sequence, an array, a map, or a function. Duplicate keys are handled as described above. The XDM representation of the inspected document is taken as value of the key-value pair.

- It is *implementation defined* if `p:json-merge` is able to process document types not mentioned yet, i.e. types of binary documents. If a processor supports a given type of documents, the key-value pair is created as described above. It is a *dynamic error* (`err:XC0107`) if a document of a not supported document type appears on port `source` of `p:json-merge`.

An implementation must bind the variable "`p:index`" in the static context of each evaluation of the XPath expression to the position of the document in the sequence of documents on port `source`, starting with "`1`".

**Document properties**

No document properties are preserved. The merged document has no `base-uri`.

## 2.18. p:label-elements

The `p:label-elements` step generates a label for each matched element and stores that label in the specified attribute.

```
<p:declare-step type="p:label-elements">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="attribute" as="xs:QName" select="'xml:id'"/>
  <p:option name="label" as="xs:string" select="'concat("_",$p:index)'"/>
  <p:option name="match" as="xs:string" select="'*'"/>
  <p:option name="replace" as="xs:boolean" select="true()"/>
</p:declare-step>
```

The value of the `label` option is an XPath expression used to generate the value of the attribute label.

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if that expression matches anything other than element nodes.

The value of the `replace` **must** be a boolean value and is used to indicate whether existing attribute values are replaced.

This step operates by generating attribute labels for each element matched. For every matched element, the expression is evaluated with the context node set to the matched element. An attribute is added to the matched element using the attribute name is specified the `attribute` option and the string value of result of evaluating the expression. If the attribute already exists on the matched element, the value is replaced with the string value only if the `replace` option has the value of `true`.

If this step is used to add or change the value of an attribute named "`xml:base`", the base URI of the element **must** also be amended accordingly.

An implementation must bind the variable "`p:index`" in the static context of each evaluation of the XPath expression to the position of the element in the sequence of matched elements. In other words, the first element (in document order) matched gets the value "`1`", the second gets the value "`2`", the third, "`3`", etc.

The result of the p:label-elements step is the input document with the attribute labels associated with matched elements. All other non-matching content remains the same.

#### Document properties

All document properties are preserved.

## 2.19. p:load

The `p:load` step has no inputs but produces as its result a document specified by an IRI.

```
<p:declare-step type="p:load">
  <p:output port="result" content-types="any"/>
  <p:option name="href" required="true" as="xs:anyURI"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="content-type" as="xs:string?"/>
  <p:option name="document-properties" as="map(xs:QName, item()*)?"/>
</p:declare-step>
```

If the option is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or the step in case of a syntactic shortcut value). If the `href` is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or `p:load` in the case of a syntactic shortcut value). It is a *dynamic error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

The document identified by the `href` URI is loaded and returned. If the URI protocol supports redirection, then redirects **must** be followed.

It is a *dynamic error* (`err:XD0011`) if the resource referenced by a `p:load` element does not exist or cannot be accessed.

The behavior of this step depends on the content type of the document loaded. The content type of a document is determined as follows:

1. If a `content-type` property is specified in `document-properties` or `content-type` is present, then the document **must** be interpreted according to that content type. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid

media type of the form "*type/subtype+ext*" or "*type/subtype*". It is a *dynamic error* (`err:XD0062`) if the `content-type` is specified and the document-properties has a "`content-type`" that is not the same.

2. If the document is retrieved with a URI protocol that specifies a content type (for example, `http:`), then the document **must** be interpreted according to that content type.

3. In the absence of an explicit type, the content type is *implementation-defined*.

The `parameters` map contains additional, optional parameters that may influence the way that content is loaded. The interpretation of this map varies according to the content type. Parameter names that are in no namespace are treated as strings using only the local-name where appropriate.

Broadly speaking, there are five categories of data that might be loaded: XML, text, JSON, HTML, and "other" binary data.

#### 2.19.1. Loading XML data

For an XML media type, the content is loaded and parsed as XML.

It is a *dynamic error* (`err:XD0049`) if the loaded content is not a well-formed XML document.

If the `dtd-validate` parameter is `true`, then DTD validation must be performed when parsing the document. It is a *dynamic error* (`err:XD0023`) if a DTD validation is performed and either the document is not valid or no DTD is found. It is a *dynamic error* (`err:XD0043`) if the `dtd-validate` parameter is `true` and the processor does not support DTD validation.

Additional XML parameters are *implementation-defined*.

#### 2.19.2. Loading text data

For a text media type, the content is loaded as a text document. (A text document is an XPath data model document consisting of a single text node.)

It is a *dynamic error* (`err:XD0060`) if the `content-type` specifies an encoding, which is not supported by the processor.

Text parameters are *implementation-defined*.

### 2.19.3. Loading JSON data

For a JSON media type, the content is loaded and parsed as JSON.

The parameters specified for the `fn:parse-json` function in [XPath and XQuery Functions and Operators 3.1] **must** be supported. Additional JSON parameters are *implementation-defined*.

It is a *dynamic error* (`err:XD0057`) if the loaded content does not conform to the JSON grammar, unless the parameter `liberal` is `true` and the processor chooses to accept the deviation.

It is a *dynamic error* (`err:XD0058`) if the parameter `duplicates` is `reject` and the value of loaded content contains a JSON object with duplicate keys.

It is a *dynamic error* (`err:XD0059`) if the parameter map contains an entry whose key is defined in the specification of `fn:parse-json` and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter `escape` with `true()` is also present.

### 2.19.4. Loading HTML data

For an HTML media type, the content is loaded and parsed into an XPath data model document that contains a tree of elements, attributes, and other nodes.

The precise way in which HTML documents are parsed into the XPath data model is *implementation-defined*.

It is a *dynamic error* (`err:XD0078`) if the loaded document cannot be represented as an HTML document in the XPath data model.

HTML parameters are *implementation-defined*.

#### 2.19.5. Loading binary data

An XProc processor may load other, arbitrary data types. How a processor interprets other media types is *implementation-defined*.

Parameters for other media types are *implementation-defined*.

#### Document properties

The properties specified in `document-properties` are applied. If the properties do not specify a `base-uri`, the `base-uri` property will reflect the base URI of the loaded document.

## 2.20. p:make-absolute-uris

The `p:make-absolute-uris` step makes an element or attribute's value in the source document an absolute IRI value in the result document.

```
<p:declare-step type="p:make-absolute-uris">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="match" required="true" as="xs:string"/>
  <p:option name="base-uri" as="xs:anyURI?"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if the pattern matches anything other than element or attribute nodes.

The value of the `base-uri` option **must** be an `anyURI`. It is interpreted as an IRI reference. If it is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or `p:make-absolute-uris` in the case of a syntactic shortcut value). It is a *dynamic error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

For every element or attribute in the input document which matches the specified pattern, its XPath string-value is resolved against the specified base URI and the resulting absolute IRI is used as the matched node's entire contents in the output.

The base URI used for resolution defaults to the matched attribute's element or the matched element's base URI unless the `base-uri` option is specified. When the `base-uri` option is specified, the option value is used as the base URI regardless of any contextual base URI value in the document. This option value is resolved against the base URI of the `p:option` element used to set the option.

If the IRI reference specified by the `base-uri` option on `p:make-absolute-uris` is absent and the input document has no base URI, the results are *implementation-dependent*.

**Document properties**

All document properties are preserved.

## 2.21. p:namespace-delete

The `p:namespace-delete` step deletes all of the namespaces identified by the specified prefixes from the document appearing on port `source`.

```
<p:declare-step type="p:namespace-delete">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="prefixes" required="true" as="xs:string"/>
</p:declare-step>
```

The value of option `prefixes` is taken as a space separated list of prefixes. It is a *dynamic error* (`err:XC0108`) if any prefix is not in-scope at the point where the `p:namespace-delete` occurs.

For any prefix the associated namespace is removed from the elements and attributes in the document appearing on port `source`. The respective elements or attributes in the document appearing on port `result` will be in no namespace.

It is a *dynamic error* (`err:XC0109`) if a namespace is to be removed from an attribute and the element already has an attribute with the resulting name.

**Document properties**

All document properties are preserved.

## 2.22. p:namespace-rename

The `p:namespace-rename` step renames any namespace declaration or use of a namespace in a document to a new IRI value.

```
<p:declare-step type="p:namespace-rename">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="from" as="xs:anyURI?"/>
  <p:option name="to" as="xs:anyURI?"/>
  <p:option name="apply-to" select="'all'"
values="('all','elements','attributes')"/>
</p:declare-step>
```

The value of the `from` option **must** be an `anyURI`. It **should** be either empty or absolute, but will not be resolved in any case.

The value of the `to` option **must** be an `anyURI`. It **should** be empty or absolute, but will not be resolved in any case.

The value of the `apply-to` option **must** be one of "`all`", "`elements`", or "`attributes`". If the value is "`elements`", only elements will be renamed, if the value is "`attributes`", only attributes will be renamed, if the value is "`all`", both elements and attributes will be renamed.

It is a *dynamic error* (`err:XC0014`) if the XML namespace (`http://www.w3.org/XML/1998/namespace`) or the XMLNS namespace (`http://www.w3.org/2000/xmlns/`) is the value of either the `from` option or the `to` option.

If the value of the `from` option is the same as the value of the `to` option, the input is reproduced unchanged on the output. Otherwise, namespace bindings, namespace attributes and element and attribute names are changed as follows:

- Namespace bindings: If the `from` option is present and its value is not the empty string, then every binding of a prefix (or the default namespace) in the input document whose value is the same as the value of the `from` option is

- replaced in the output with a binding to the value of the `to` option, provided it is present and not the empty string;
- otherwise (the `to` option is not specified or has an empty string as its value) absent from the output.

If the `from` option is absent, or its value is the empty string, then no bindings are changed or removed.

- Elements and attributes: If the `from` option is present and its value is not the empty string, for every element and attribute, as appropriate, in the input whose namespace name is the same as the value of the `from` option, in the output its namespace name is
  - replaced with the value of the `to` option, provided it is present and not the empty string;
  - otherwise (the `to` option is not specified or has an empty string as its value) changed to have no value.

  If the `from` option is absent, or its value is the empty string, then for every element and attribute, as appropriate, whose namespace name has no value, in the output its namespace name is set to the value of the `to` option.

  It is a *dynamic error* (`err:XC0092`) if as a consequence of changing or removing the namespace of an attribute the attribute's name is not unique on the respective element.
- Namespace attributes: If the `from` option is present and its value is not the empty string, for every namespace attribute in the input whose value is the same as the value of the `from` option, in the output
  - the namespace attribute's value is replaced with the value of the `to` option, provided it is present and not the empty string;
  - otherwise (the `to` option is not specified or has an empty string as its value) the namespace attribute is absent.

> **Note**
>
> The `apply-to` option is primarily intended to make it possible to avoid renaming attributes when the `from` option specifies no namespace, since many attributes are in no namespace.
>
> Care should be taken when specifying no namespace with the `to` option. Prefixed names in content, for example QNames and XPath expressions, may end up with no appropriate namespace binding.

**Document properties**

All document properties are preserved.

## 2.23. p:pack

The `p:pack` step merges two document sequences in a pair-wise fashion.

```
<p:declare-step type="p:pack">
  <p:input port="source" content-types="text xml html" sequence="true"
primary="true"/>
  <p:input port="alternate" sequence="true" content-types="text xml html"/>
  <p:output port="result" sequence="true" content-types="application/xml"/>
  <p:option name="wrapper" required="true" as="xs:QName"/>
</p:declare-step>
```

The step takes each pair of documents, in order, one from the `source` port and one from the `alternate` port, wraps them with a new element node whose QName is the value specified in the `wrapper` option, and writes that element to the `result` port as a document.

If the step reaches the end of one input sequence before the other, then it simply wraps each of the remaining documents in the longer sequence.

> **Note**
>
> In the common case, where the document element of a document in the `result` sequence has two element children, any comments, processing instructions, or white space text nodes that occur between them may have come from either of the input documents; this step does not attempt to distinguish which one.

**Document properties**

No document properties are preserved. The result documents do not have a `base-uri` property.

## 2.24. p:rename

The `p:rename` step renames elements, attributes, or processing-instruction targets in a document.

```
<p:declare-step type="p:rename">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
  <p:option name="new-name" required="true" as="xs:QName"/>
</p:declare-step>
```

The value of the `match` option must be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if the pattern matches anything other than element, attribute or processing instruction nodes.

Each element, attribute, or processing-instruction in the input matched by the *selection pattern* specified in the `match` option is renamed in the output to the name specified by the `new-name` option.

If the `match` option matches an attribute and if the element on which it occurs already has an attribute whose expanded name is the same as the expanded name of

the specified `new-name`, then the results is as if the current attribute named "*new-name*" was deleted before renaming the matched attribute.

With respect to attributes named "`xml:base`", the following semantics apply: renaming an *from* "`xml:base`" *to* something else has no effect on the underlying base URI of the element; however, if an attribute is renamed *from* something else *to* "`xml:base`", the base URI of the element **must** also be amended accordingly.

If the pattern matches processing instructions, then it is the processing instruction target that is renamed. It is a *dynamic error* (`err:XC0013`) if the pattern matches a processing instruction and the new name has a non-null namespace.

#### Document properties

All document properties are preserved.

## 2.25. p:replace

The `p:replace` step replaces matching nodes in its primary input with the top-level node(s) of the `replacement` port's document.

```
<p:declare-step type="p:replace">
  <p:input port="source" primary="true" content-types="xml html"/>
  <p:input port="replacement" content-types="text xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="match" required="true" as="xs:string"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if that pattern matches an attribute or a namespace nodes. Multiple matches are allowed, in which case multiple copies of the `replacement` document will occur.

Every node in the primary input matching the specified pattern is replaced in the output by the top-level node(s) of the `replacement` document. Only non-nested matches are replaced. That is, once a node is replaced, its descendants cannot be matched.

If the document node is matched and port `replacement` contains a text document, the entire document is replaced by the text node. What appears on port `result` is a text document with the text node wrapped in a document node.

#### Document properties

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. For other document types, all document properties are preserved.

## 2.26. p:set-attributes

The `p:set-attributes` step sets attributes on matching elements.

```
<p:declare-step type="p:set-attributes">
  <p:input port="source" primary="true" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
  <p:option name="attributes" required="true" as="map(xs:QName,
xs:anyAtomicType)"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if that pattern matches anything other than element nodes.

A new attribute is created for each entry in the map appearing on the `attributes` option. The attribute name is taken from the entry's key while the attribute value is taken from the string value of the entry's value.

If an attribute with the same name as one of the attributes to be created already exists, the value specified on the `attributes` option is used. The result port of this step produces a copy of the `source` port's document with the matching elements' attributes modified.

The matching elements are specified by the *selection pattern* in the `match` option. All matching elements are processed. If no elements match, the step will not change any elements.

If the attributes taken from the `attributes` use namespaces, prefixes, or prefixes bound to different namespaces, the document produced on the `result` output port will require namespace fixup.

If an attribute named `xml:base` is added or changed, the base URI of the element **must** also be amended accordingly.

#### Document properties

All document properties are preserved.

## 2.27. p:set-properties

The `p:set-properties` step sets document properties on the source document.

```
<p:declare-step type="p:set-properties">
  <p:input port="source" content-types="any"/>
  <p:output port="result" content-types="any"/>
  <p:option name="properties" required="true" as="map(xs:QName,item()*)"/>
  <p:option name="merge" select="true()" as="xs:boolean"/>
</p:declare-step>
```

The document properties of the document on the `source` port are augmented with the values specified in the `properties` option. The document produced on the `result` port has the same representation but the adjusted property values.

If the `merge` option is true, then the supplied properties are added to the existing properties, overwriting already existing values for a given key. If it is false, the document's properties are replaced by the new set.

It is a *dynamic error* (`err:XD0070`) if a value is assigned to the `serialization` document property that cannot be converted into `map(xs:QName, item()*)` according to the rules in section "QName handling" of [XProc 3.0].

It is a *dynamic error* (`err:XC0069`) if the `properties` map contains a key equal to the string "`content-type`".

If the `properties` map contains a key equal to the string "`base-uri`" the associated value is taken as the new base URI of the resulting document. It is a *dynamic*

*error* (`err:XD0064`) if the base URI is not both absolute and valid according to [RFC 3986].

#### Document properties

If `merge` is true, document properties not overridden by settings in the `properties` map are preserved, otherwise the resulting document has only the `content-type` property and the properties specified in the `properties` map. In particular, if `merge` is false, the `base-uri` property will not be preserved. This means that the resulting document will not have a base URI if the `properties` map does not contain a `base-uri` entry.

## 2.28. p:sink

The `p:sink` step accepts a sequence of documents and discards them. It has no output.

```
<p:declare-step type="p:sink">
  <p:input port="source" content-types="any" sequence="true"/>
</p:declare-step>
```

#### Document properties

Not applicable.

## 2.29. p:split-sequence

The `p:split-sequence` step accepts a sequence of documents and divides it into two sequences.

```
<p:declare-step type="p:split-sequence">
  <p:input port="source" content-types="any" sequence="true"/>
  <p:output port="matched" sequence="true" primary="true" content-
types="any"/>
  <p:output port="not-matched" sequence="true" content-types="any"/>
  <p:option name="initial-only" as="xs:boolean" select="false()"/>
  <p:option name="test" required="true" as="xs:string"/>
</p:declare-step>
```

The value of the `test` option **must** be an XPathExpression.

The XPath expression in the `test` option is applied to each document in the input sequence. If the effective boolean value of the expression is true, the document is copied to the `matched` port; otherwise it is copied to the `not-matched` port.

If the `initial-only` option is true, then when the first document that does not satisfy the test expression is encountered, it *and all the documents that follow it* are written to the `not-matched` port. In other words, it only writes the initial series of matched documents (which may be empty) to the `matched` port. All other documents are written to the `not-matched` port, irrespective of whether or not they match.

The XPath context for the `test` option changes over time. For each document that appears on the `source` port, the expression is evaluated with that document as the context document. The context position (`position()`) is the position of that document within the sequence and the context size (`last()`) is the total number of documents in the sequence. It is a *dynamic error* (`err:XC0150`) if evaluating the XPath expression in option `test` on a context document results in an error.

> **Note**
>
> In principle, this component cannot stream because it must buffer all of the input sequence in order to find the context size. In practice, if the test expression does not use the `last()` function, the implementation can stream and ignore the context size.

If the implementation supports passing PSVI annotations between steps, the `p:split-sequence` step **must** preserve any annotations that appear in the input.

**Document properties**

All document properties are preserved.

## 2.30. p:store

The `p:store` step stores (a possibly serialized version of) its input to a URI. It outputs a reference to the location of the stored document on the `result-uri` port. Aside from these side-effects, it behaves like a `p:identity` step, copying its input to the `result` port.

```
<p:declare-step type="p:store">
  <p:input port="source" content-types="any"/>
  <p:output port="result" content-types="any" primary="true"/>
  <p:output port="result-uri" content-types="application/xml"/>
  <p:option name="href" required="true" as="xs:anyURI"/>
  <p:option name="serialization" as="map(xs:QName,item()*)?"/>
</p:declare-step>
```

The value of the `href` option **must** be an `anyURI`. If it is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or `p:store` in the case of a syntactic shortcut value).

The step attempts to store the document to the specified URI. If the URI scheme "`file:`" is supported, the processor **should** try to create all non existing folders in the URI's path. It is a *dynamic error* (`err:XC0050`) if the URI scheme is not supported or the step cannot store to the specified location.

The output of this step on the `result-uri` port is a document containing a single `c:result` element whose content is the absolute URI of the document stored by the step.

The `serialization` option is provided to control the serialization of content when it is stored. If the document to be stored has a "serialization" property, the serialization is controlled by the merger of the two maps where the entries in the "serialization" property take precedence. Serialization is described in [XProc 3.0].

**Document properties**

All document properties are preserved.

## 2.31. p:string-replace

The `p:string-replace` step matches nodes in the document provided on the `source` port and replaces them with the string result of evaluating an XPath expression.

```
<p:declare-step type="p:string-replace">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="match" required="true" as="xs:string"/>
  <p:option name="replace" required="true" as="xs:string"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern.

The value of the `replace` option **must** be an XPathExpression.

The matched nodes are specified with the *selection pattern* in the `match` option. For each matching node, the XPath expression provided by the `replace` option is evaluated with the matching node as the XPath context node. The string value of the result is used in the output. Nodes that do not match are copied without change.

If the expression given in the `match` option matches an *attribute*, the string value of the `replace` expression is used as the new value of the attribute in the output. If the attribute is named "`xml:base`", the base URI of the element **must** also be amended accordingly.

If the document node is matched, the entire document is replaced by the string value of the `replace` expression. What appears on port `result` is a text document with the text node wrapped in a document node.

If the expression matches any other kind of node, the entire node (and *not* just its contents) is replaced by the string value of the `replace` expression.

**Document properties**

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. For other document types, all document properties are preserved.

## 2.32. p:text-count

The `p:text-count` step counts the number of lines in a text document and returns a single XML document containing that number.

```
<p:declare-step type="p:text-count">
  <p:input port="source" primary="true" sequence="false" content-
types="text"/>
  <p:output port="result" primary="true" sequence="false" content-
types="application/xml"/>
</p:declare-step>
```

The `p:text-count` step counts the number of lines in the text document appearing on its `source` port. It returns on its `result` port an XML document containing a single `c:result` element whose contents is the string representing this count.

Lines are identified as described in XML, 2.11 End-of-Line Handling. For the purpose of identifying lines, if the very last character in the text document is a newline (
), that newline is ignored. (It is not a separator between that line and a following line that contains no characters.)

**Document properties**

No document properties are preserved. The count document does not have a `base-uri` property.

## 2.33. p:text-head

The `p:text-head` step returns lines from the beginning of a text document.

```
<p:declare-step type="p:text-head">
  <p:input port="source" primary="true" sequence="false" content-
types="text"/>
  <p:output port="result" primary="true" sequence="false" content-
types="text"/>
  <p:option name="count" required="true" as="xs:integer"/>
</p:declare-step>
```

The `p:text-head` step returns on its `result` port lines from the text document that appears on its `source` port:

- If the `count` option is positive, the `p:text-head` step returns the first `count` lines
- If the `count` option is zero, the `p:text-head` step returns all lines
- If the `count` option is negative, the `p:text-head` step returns all lines except the first `count` lines

Lines are identified as described in XML, 2.11 End-of-Line Handling. All lines returned by `p:text-head` are terminated with a single newline (
).

#### Document properties

All document properties are preserved.

## 2.34. p:text-join

The `p:text-join` step concatenates text documents.

```
<p:declare-step type="p:text-join">
  <p:input port="source" sequence="true" content-types="text"/>
  <p:output port="result" content-types="text"/>
  <p:option name="separator" as="xs:string?"/>
  <p:option name="prefix" as="xs:string?"/>
  <p:option name="suffix" as="xs:string?"/>
  <p:option name="override-content-type" as="xs:string?"/>
</p:declare-step>
```

The `p:text-join` step concatenates the text documents appearing on its `source` port into a single document on its `result` port. The documents will be concatenated in order of appearance.

- When the `separator` option is specified, its value will be inserted in between adjacent documents.
- When the `prefix` option is specified, the document appearing on the `result` port will always start with its value (also when there are no documents on the `source` port).
- When the `suffix` option is specified, the document appearing on the `result` port will always end with its value (also when there are no documents on the `source` port).

When the `override-content-type` option is specified, the document appearing on the port `result` will have this media type as part of its document properties. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*`type/subtype+ext`*" or "*`type/subtype`*". It is a *dynamic error* (`err:XC0001`) if the value of option `override-content-type` is not a text media type.

Concatenating text documents does not require identifying individual lines in each document, consequently no special end-of-line handling is performed.

#### Document properties

No document properties are preserved. The joined document has no `base-uri` property.

## 2.35. p:text-replace

The `p:text-replace` step replaces all occurrences of substrings in a text document that match a supplied regular expression with a given replacement string.

```
<p:declare-step type="p:text-replace">
  <p:input port="source" primary="true" sequence="false" content-
types="text"/>
  <p:output port="result" primary="true" sequence="false" content-
types="text"/>
  <p:option name="pattern" required="true" as="xs:string"/>
  <p:option name="replacement" required="true" as="xs:string"/>
  <p:option name="flags" as="xs:string?"/>
</p:declare-step>
```

The `p:text-replace` step replaces all occurrences of substrings in the text document appearing on its `source` port that match a supplied regular expression with a given replacement string. The result is returned (as another text document) on its `result` port.

This step is a convenience wrapper around the XPath `fn:replace` function to ease text replacements in the document flow of a pipeline.

The `pattern`, `replacement` and `flags` options are specified the same as the parameters with the same names of the `fn:replace` function. The `pattern` option **must** be a regular expression as specified in [XPath and XQuery Functions and Operators 3.1], section 7.61 "`Regular Expression Syntax`". It is a *dynamic error* (`err:XC0147`) if the specified value is not a valid XPath regular expression.

Replacing strings in text documents does not require identifying individual lines in each document, consequently no special end-of-line handling is performed.

#### Document properties

All document properties are preserved.

## 2.36. p:text-sort

The `p:text-sort` step sorts lines in a text document.

```
<p:declare-step type="p:text-sort">
  <p:input port="source" primary="true" sequence="false" content-
types="text"/>
  <p:output port="result" primary="true" sequence="false" content-
types="text"/>
  <p:option name="sort-key" as="xs:string" select="'.'"/>
  <p:option name="order" as="xs:string" select="'ascending'"
values="('ascending', 'descending')"/>
  <p:option name="case-order" as="xs:string?" values="('upper-first',
'lower-first')"/>
  <p:option name="lang" as="xs:language?"/>
  <p:option name="collation" as="xs:string" select="'https://www.w3.org/
2005/xpath-functions/collation/codepoint'"/>
  <p:option name="stable" as="xs:boolean" select="true()"/>
</p:declare-step>
```

The `p:text-sort` step sorts the lines in the text document appearing on its `source` port and returns the result as another text document on its `result` port. The sort key is obtained by applying the XPath expression in `sort-key` to each line in turn.

- The `sort-key` is used to obtain a sort key for each of the lines in the document appearing on `source`. The `context item` is the line as an instance of `xs:string`, the `context position` is the number of the line in the document on port `source`, the `context size` is the number of lines in this document. It is a *dynamic error* (`err:XC0098`) if a dynamic XPath error occurred while applying sort-key to a line. It is a *dynamic error* (`err:XC0099`) if the result of applying `sort-key` to a given line results in a sequence with more than one item.

- The `order` option defines whether the lines are processed in ascending or descending order. Its value **must** be one of `ascending` or `descending`. The default is `ascending`.

- The `case-order` option defines whether upper-case letters are to be collated before or after lower-case letters. Its value **must** be one of `upper-first` or `lower-first`. The default is language-dependent.

- The `lang` option defines the language whose collating conventions are to be used. The default depends on the processing environment. Its value must be a valid language code (e.g. `en-EN`).

- The `collation` option identifies how strings are to be compared with each other. Its value must be a valid collation URI. The only collation XProc processors **must** support is the Unicode Codepoint Collation `http://www.w3.org/2005/xpath-functions/collation/codepoint`. This is also its default. Support for other collations is *implementation-defined*.

- If the `stable` option is set to `false` this indicates that there is no requirement to retain the original order of items that have equal values for all the sort keys.

Lines are identified as described in XML, 2.11 End-of-Line Handling. For the purpose of identifying lines, if the very last character in the text document is a newline (
), that newline is ignored. (It is not a separator between that line and a following line that contains no characters.) All lines returned by `p:text-sort` are terminated with a single newline (
).

The sort process performed by this step is the same as described in The xsl:sort Element. Options `lang` and `case-order` are only taken into consideration if no value is selected for option `collation`.

**Document properties**

All document properties are preserved.

## 2.37. p:text-tail

The `p:text-tail` step returns lines from the end of a text document.

```
<p:declare-step type="p:text-tail">
  <p:input port="source" primary="true" sequence="false" content-
types="text"/>
  <p:output port="result" primary="true" sequence="false" content-
types="text"/>
  <p:option name="count" required="true" as="xs:integer"/>
</p:declare-step>
```

The `p:text-tail` step returns on its `result` port lines from the text document that appears on its `source` port:

- If the `count` option is positive, the `p:text-tail` step returns the last `count` lines
- If the `count` option is zero, the `p:text-tail` step returns all lines
- If the `count` option is negative, the `p:text-tail` step returns all lines except the last `count` lines

Lines are identified as described in XML, 2.11 End-of-Line Handling. All lines returned by `p:text-tail` are terminated with a single newline (
).

**Document properties**

All document properties are preserved.

## 2.38. p:unarchive

The `p:unarchive` step outputs on its `result` port specific entries in an archive (for instance from a zip file).

```
<p:declare-step type="p:unarchive">
  <p:input port="source" primary="true" content-types="any"
sequence="false"/>
  <p:output port="result" primary="true" content-types="any"
sequence="true"/>
  <p:option name="include-filter" as="xs:string*"/>
  <p:option name="exclude-filter" as="xs:string*"/>
  <p:option name="format" as="xs:QName?"/>
  <p:option name="parameters" as="map(xs:QName, item()*)?"/>
  <p:option name="relative-to" as="xs:anyURI?"/>
  <p:option name="override-content-types" as="array(array(xs:string))?"/>
</p:declare-step>
```

The meaning and interpretation of the `p:unarchive` step's options is as follows:

- The format of the archive is determined as follows:
  - If the `format` option is specified, this determines the format of the archive. Implementations **must** support the [ZIP] format, specified with the value `zip`. It is *implementation-defined* what other formats are supported.
  - If no `format` option is specified or if its value is the empty sequence, the archive's format will be determined by the step, using the `content-type` document-property of the document on the `source` port and/or by inspecting its contents. It is *implementation-defined* how the step determines the archive's format. Implementations **should** recognize archives in [ZIP] format.
  - It is a *dynamic error* (`err:XC0085`) if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.
- The `parameters` option can be used to supply parameters to control the unarchiving. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

- If present, the value of the `include-filter` or `exclude-filter` option **must** be a sequence of strings, each one representing a regular expressions as specified in [XPath and XQuery Functions and Operators 3.1], section 7.61 "`Regular Expression Syntax`". It is a *dynamic error* (`err:XC0147`) if a specified value is not a valid XPath regular expression.

  If neither the `include-filter` option nor the `exclude-filter` option is specified, the `p:unarchive` step outputs on its `result` port all entries in the archive.

  If the `include-filter` option or the `exclude-filter` option is specified, the `p:archive` step outputs on the `result` port the entries from the archive that conform to the following rules:

  - If any `include-filter` pattern matches an archive entry's name, the entry is included in the output.
  - If any `exclude-filter` pattern matches an archive entry's name, the entry is excluded in the output.
  - If both options are provided, the include filter is processed first, then the exclude filter.
  - Names of entries in archives are always relative names. For instance, the name of a file called `xyz.xml` in a `specs` subdirectory in an archive is called in full `specs/xyz.xml` (and not `/specs/xyz.xml`).

  As a result: an item is included if it matches (at least) one of the `include-filter` values and none of the `exclude-filter` values.

  The regular expressions specified in the `include-filter` and `exclude-filter` options will be matched against the path of the entry *in* the archive. The matching is done unanchored: it is a match if the regular expression matches part of the entry's path. Informally: matching behaves like applying the XPath `matches#2` function, like in `matches($path-in-archive, $regular-expression)`.

> **Note**
>
> Depending on how archives are constructed, the path of an entry in an archive can be with or without a leading slash. Usually it will be without. For archives constructed by `p:archive` no leading slash will be present.

- The `relative-to` option, when present, is used in creating the base URI of the unarchived documents. If the option is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or the step in case of a syntactic shortcut value).
- The `override-content-types` option can be used to partially override the content-type determination mechanism, as described in Section 2.4.1, "Overriding content types".

The base URI of an unarchived document appearing on the `result` port is:

- If the `relative-to` option is present: Function `p:urify()` is called with the value of this option as second parameter (`$basedir`) and with the relative path of this document as it was in the archive as first parameter
- If the `relative-to` option is *not* present: Function `p:urify()`is called with the value of the base URI of the archive appended with a "`/`" as second parameter (`$baseDir`) and the relative path of this document as it was in the archive as first parameter

It is a *dynamic error* (`err:XC0120`) if the `relative-to` option is not present and the document on the `source` port does not have a base URI. It is a *dynamic error* (`err:XD0064`) if the option is not a valid URI according to [RFC 3986].

For instance, the base URI of an unarchived file called `xyz.xml` that resided in the `specs` subdirectory in an archive with base URI `file:///a/b/c.zip` will become:

- With the `relative-to` option set to `file:///x/y/z`: `file:///x/y/z/specs/xyz.xml`

- Without a `relative-to` option set: `file:///a/b/c.zip/specs/xyz.xml`

#### Document properties

No document properties are preserved. The `base-uri` property of each unarchived document is reflective of the base URI of the document.

## 2.39. p:uncompress

The `p:uncompress` step expects on its `source` port a compressed document. It outputs an uncompressed version of this on its `result` port.

```
<p:declare-step type="p:uncompress">
  <p:input port="source" primary="true" content-types="any"
sequence="false"/>
  <p:output port="result" primary="true" content-types="any"
sequence="false"/>
  <p:option name="format" as="xs:QName?"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="content-type" as="xs:string" select="'application/octet-
stream'"/>
</p:declare-step>
```

The compression format of the document appearing on the `source` port is determined as follows:

- If the `format` option is specified, this determines the compression format. Implementations **must** support the [GZIP] format, specified with the value `gzip`. It is *implementation-defined* what other formats are supported. It is a *dynamic error* (`err:XC0202`) if the compression format cannot be understood, determined and/or processed.

- If no `format` option is specified or its value is the empty sequence, the compression format will be determined by the step, using the `content-type` document-property of the document on the `source` port and/or by inspecting its contents. It is *implementation-defined* how the step determines the compression format. Implementations **should** recognize archives in [GZIP] format.

The `parameters` option can be used to supply parameters to control the uncompression. The semantics of the keys and the allowed values for these keys are *implementation-defined*. It is a *dynamic error* (`err:XC0079`) if the map `parameters` contains an entry whose key is defined by the implementation and whose value is not valid for that key.

Identification of the uncompressed document's content-type is done as follows:

1. If the `content-type` option is specified, the uncompressed document **must** be interpreted according to that content-type. It is a *dynamic error* (`err:XD0079`) if a supplied content-type is not a valid media type of the form "*`type/subtype+ext`*" or "*`type/subtype`*". It is a *dynamic error* (`err:XC0201`) if the `p:uncompress` step cannot perform the requested content-type cast.

2. In the absence of an explicit type, the content will be interpreted as content type `application/octet-stream`.

#### Document properties

All document properties are preserved, except for the `content-type` property which is updated accordingly.

## 2.40. p:unwrap

The `p:unwrap` step replaces matched elements with their children.

```
<p:declare-step type="p:unwrap">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if that pattern matches anything other than the document node or element nodes.

Every element in the `source` document that matches the specified `match` pattern is replaced by its children, effectively "unwrapping" the children from their parent. Non-element nodes and unmatched elements are passed through unchanged.

> **Note**
>
> The matching applies to the entire document, not just the "top-most" matches. A pattern of the form `h:div` will replace *all* `h:div` elements, not just the top-most ones.

This step produces a single document. Special cases:

- If the document element is unwrapped, the result might not be well-formed XML.

  For instance unwrapping the root element of `<!-- COMMENT --><root-element/>` will result in a document node with a single comment node child, which is not well-formed.

- If a document consisting of only an empty root element is unwrapped, the result will be a document node without children. The result document's content type will not change.

- If a document consisting of a root element containing only text is unwrapped, the result will be a document node with a single text node child. The result document's content type will become "`text/plain`".

As specified in the core language specification: if the content type changes, the `serialization` document property, if present, will be removed.

**Document properties**

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. In all other cases, all document properties are preserved.

## 2.41. p:uuid

The `p:uuid` step generates a [UUID] and injects it into the `source` document.

```
<p:declare-step type="p:uuid">
  <p:input port="source" primary="true" content-types="xml html"/>
  <p:output port="result" content-types="text xml html"/>
  <p:option name="match" as="xs:string" select="'/*'"/>
  <p:option name="version" as="xs:integer?"/>
</p:declare-step>
```

The value of the `match` option must be an XSLTSelectionPattern. The value of the `version` option must be an integer.

If the `version` is specified, that version of UUID must be computed. It is a *dynamic error* (`err:XC0060`) if the processor does not support the specified `version` of the UUID algorithm. If the `version` is not specified, the version of UUID computed is *implementation-defined*.

Implementations **must** support version 4 UUIDs. Support for other versions of UUID, and the mechanism by which the necessary inputs are made available for computing other versions, is *implementation-defined*.

The matched nodes are specified with the *selection pattern* in the `match` option. For each matching node, the generated UUID is used in the output (if more than one node matches, the *same* UUID is used in each match). Nodes that do not match are copied without change.

If the expression given in the `match` option matches an *attribute*, the UUID is used as the new value of the attribute in the output. If the attribute is named "`xml:base`", the base URI of the element **must** also be amended accordingly.

If the document node is matched, the entire document is replaced by a text node with the UUID. What appears on port `result` is a text document with the text node wrapped in a document node.

If the expression matches any other kind of node, the entire node (and *not* just its contents) is replaced by the UUID.

**Document properties**

If the resulting document contains exactly one text node, the `content-type` property is changed to `text/plain` and the `serialization` property is removed, while all other document properties are preserved. For other document types, all document properties are preserved.

## 2.42. p:wrap-sequence

The `p:wrap-sequence` step accepts a sequence of documents and produces either a single document or a new sequence of documents.

```
<p:declare-step type="p:wrap-sequence">
  <p:input port="source" content-types="text xml html" sequence="true"/>
  <p:output port="result" sequence="true" content-types="application/xml"/>
  <p:option name="wrapper" required="true" as="xs:QName"/>
  <p:option name="group-adjacent" as="xs:string?"/>
</p:declare-step>
```

The value of the `group-adjacent` option **must** be an XPathExpression.

In its simplest form, `p:wrap-sequence` takes a sequence of documents and produces a single, new document by placing each document in the `source` sequence inside a new document element as sequential siblings. The name of the document element is the value specified in the `wrapper` option.

The `group-adjacent` option can be used to group adjacent documents. The XPath context for the `group-adjacent` option changes over time. For each document that appears on the `source` port, the expression is evaluated with that document as the context document. The context position (`position()`) is the position of that document within the sequence and the context size (`last()`) is the total number of documents in the sequence. Whenever two or more sequentially adjacent documents have the same “group adjacent” value, they are wrapped together in a single wrapper element. Two “group adjacent” values are the same if the standard XPath function `deep-equal()` returns true for them.

**Document properties**

No document properties are preserved. The document produced has no `base-uri` property.

## 2.43. p:wrap

The `p:wrap` step wraps matching nodes in the `source` document with a new parent element.

```
<p:declare-step type="p:wrap">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="application/xml"/>
  <p:option name="wrapper" required="true" as="xs:QName"/>
  <p:option name="match" required="true" as="xs:string"/>
  <p:option name="group-adjacent" as="xs:string?"/>
</p:declare-step>
```

The value of the `match` option **must** be an XSLTSelectionPattern. It is a *dynamic error* (`err:XC0023`) if the pattern matches anything other than document, element, text, processing instruction, and comment nodes.

The value of the `group-adjacent` option **must** be an XPathExpression.

If the node matched is the document node (`match="/"`), the result is a new document where the document element is a new element node whose QName is the value specified in the `wrapper` option. That new element contains copies of all of the children of the original document node.

When the *selection pattern* does not match the document node, every node that matches the specified `match` pattern is replaced with a new element node whose QName is the value specified in the `wrapper` option. The content of that new element is a copy of the original, matching node. The `p:wrap` step performs a "deep" wrapping, the children of the matching node and their descendants are processed and wrappers are added to all matching nodes.

The `group-adjacent` option can be used to group adjacent matching nodes in a single wrapper element. The specified XPath expression is evaluated for each matching node with that node as the XPath context node. Whenever two or more adjacent matching nodes have the same “group adjacent” value, they are wrapped

together in a single wrapper element. Two "group adjacent" values are the same if the standard XPath function `deep-equal()` returns true for them.

Two matching nodes are considered adjacent if and only if they are siblings and either there are no nodes between them or all intervening, non-matching nodes are whitespace text, comment, or processing instruction nodes.

#### Document properties

All document properties are preserved.

## 2.44. p:www-form-urldecode

The `p:www-form-urldecode` step decodes a `x-www-form-urlencoded` string into a JSON representation.

```
<p:declare-step type="p:www-form-urldecode">
  <p:output port="result" content-types="application/json"/>
  <p:option name="value" required="true" as="xs:string"/>
</p:declare-step>
```

A JSON object of the form "`map(xs:string, xs:string+)`" will appear on `result` port. The `value` option is interpreted as a string of parameter values encoded using the `x-www-form-urlencoded` algorithm. Each name/value pair is represented in the JSON object as key/value entry.

It is a *dynamic error* (`err:XC0037`) if the `value` provided is not a properly `x-www-form-urlencoded` value.

If any parameter name occurs more than once in the encoded string, a sequence will be associated with the respective key. The order in the sequence retains the order of name/value pairs in the encoded string.

#### Document properties

The resulting JSON document has no properties apart from `content-type`. In particular, it has no `base-uri`.

## 2.45. p:www-form-urlencode

The `p:www-form-urlencode` step encodes a set of parameter values as a `x-www-form-urlencoded` string.

```
<p:declare-step type="p:www-form-urlencode">
  <p:output port="result" content-types="text/plain"/>
  <p:option name="parameters" required="true"
as="map(xs:string,xs:anyAtomicType+)"/>
</p:declare-step>
```

The map entries of `parameters` option are encoded as a single `x-www-form-urlencoded` string of name/value pairs. This string is returned on the `result` port as a text document.

If more than one value is associated with a given key in `parameters` option, a name/value pair is created for each value.

#### Document properties

The resulting text document has no properties apart from `content-type`. In particular, it has no `base-uri`.

## 2.46. p:xinclude

The `p:xinclude` step applies [XInclude] processing to the `source` document.

```
<p:declare-step type="p:xinclude">
  <p:input port="source" content-types="xml html"/>
  <p:output port="result" content-types="xml html"/>
  <p:option name="fixup-xml-base" as="xs:boolean" select="false()"/>
  <p:option name="fixup-xml-lang" as="xs:boolean" select="false()"/>
</p:declare-step>
```

The value of the `fixup-xml-base` option **must** be a boolean. If it is true, base URI fixup will be performed as per [XInclude].

The value of the `fixup-xml-lang` option **must** be a boolean. If it is true, language fixup will be performed as per [XInclude].

The included documents are located with the base URI of the input document and are not provided as input to the step.

It is a *dynamic error* (`err:XC0029`) if an XInclude error occurs during processing.

**Document properties**

All document properties are preserved.

## 2.47. p:xquery

The `p:xquery` step applies an XQuery query to the sequence of documents provided on the `source` port.

```
<p:declare-step type="p:xquery">
  <p:input port="source" content-types="any" sequence="true"
primary="true"/>
  <p:input port="query" content-types="text xml"/>
  <p:output port="result" sequence="true" content-types="any"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="version" as="xs:string?"/>
</p:declare-step>
```

If a sequence of documents is provided on the `source` port, the first document is used as the initial context item. The whole sequence is also the default collection. If no documents are provided on the `source` port, the initial context item is undefined and the default collection is empty.

The `query` port must receive a single document which is either an XML document or a text document. A text document **must** be treated as the query. For an XML document the following rules apply:

- If the document root element is `c:query`, the text descendants of this element are considered the query.

```
<c:query>
    string
</c:query>
```

- If the document root element is in the XQueryX namespace, the document is treated as an XQueryX-encoded query. Support for XQueryX is *implementation-defined*.
- Otherwise the serialization of the document **must** be treated as the query. The document's serialization property (if present) is used.

If the step specifies a `version`, then that version of XQuery **must** be used to process the transformation. It is a *dynamic error* (`err:XC0009`) if the specified XQuery version is not available. If the step does not specify a version, the implementation may use any version it has available and may use any means to determine what version to use, including, but not limited to, examining the version of the query.It is *implementation defined* which XQuery version(s) is/are supported.

The name/value pairs in option `parameters` are used to set the query's external variables.

It is a *dynamic error* (`err:XC0101`) if a document appearing on port `source` cannot be represented in the XDM version associated with the chosen XQuery version, e.g. when a JSON document contains a map and XDM 3.0 is used. It is a *dynamic error* (`err:XC0102`) if any key in option `parameters` is associated to a value that cannot be represented in the XDM version associated with the chosen XQuery version, e.g. with a map, an array, or a function when XDM 3.0 is used.

It is a *dynamic error* (`err:XC0103`) if any error occurs during XQuery's static analysis phase. It is a *dynamic error* (`err:XC0104`) if any error occurs during XQuery's dynamic evaluation phase.

The output of this step **may** include PSVI annotations.

The static context of the XQuery processor is augmented in the following way:

**Statically known default collection type**
`document()*`

**Statically known namespaces:**
Unchanged from the implementation defaults. No namespace declarations in the XProc pipeline are automatically exposed in the static context.

The dynamic context of the XQuery processor is augmented in the following way:

**Context item**
The first document that appears on the `source` port.

**Context position**
1

**Context size**
1

**Variable values**
Any parameters passed in the `parameters` option augment any implementation-defined variable bindings known to the XQuery processor.

**Function implementations**
The function implementations provided by the XQuery processor.

**Current dateTime**
The point in time returned as the current dateTime is *implementation-defined*.

**Implicit timezone**
The implicit timezone is *implementation-defined*.

**Available documents**
The set of available documents (those that may be retrieved with a URI) is *implementation-dependent*.

**Available collections**
The set of available collections is *implementation-dependent*.

**Default collection**
The sequence of documents provided on the `source` port.

#### 2.47.1. Example

The following pipeline applies XInclude processing and schema validation before using XQuery:

Example 1. A Sample Pipeline Document

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                version="3.0">
<p:input port="source"/>
<p:output port="result"/>

<p:xinclude/>

<p:validate-with-xml-schema name="validate">
  <p:with-input port="schema"
                href="http://example.com/path/to/schema.xsd"/>
</p:validate-with-xml-schema>

<p:xquery>
   <p:with-input port="query" href="countp.xq"/>
</p:xquery>

</p:declare-step>
```

Where `countp.xq` might contain:

```
<count>{count(.//p)}</count>
```

#### 2.47.2. Document properties

No document properties are preserved. The `base-uri` property of each document will reflect the base URI specified by the query. If the query does not establish a base URI, the document will not have one.

## 2.48. p:xslt

The `p:xslt` step invokes an XSLT stylesheet.

```
<p:declare-step type="p:xslt">
  <p:input port="source" content-types="any" sequence="true"
primary="true"/>
  <p:input port="stylesheet" content-types="xml"/>
  <p:output port="result" primary="true" sequence="true" content-
types="any"/>
  <p:output port="secondary" sequence="true" content-types="any"/>
  <p:option name="parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="static-parameters" as="map(xs:QName,item()*)?"/>
  <p:option name="global-context-item" as="item()?"/>
  <p:option name="populate-default-collection" as="xs:boolean?"
select="true()"/>
  <p:option name="initial-mode" as="xs:QName?"/>
  <p:option name="template-name" as="xs:QName?"/>
  <p:option name="output-base-uri" as="xs:anyURI?"/>
  <p:option name="version" as="xs:string?"/>
</p:declare-step>
```

If `output-base-uri` is relative, it is made absolute against the base URI of the element on which it is specified (`p:with-option` or `p:xslt` in the case of a syntactic shortcut value).

If the step specifies a `version`, then that version of XSLT **must** be used to process the transformation. It is a *dynamic error* (`err:XC0038`) if the specified xslt version is not available. If the step does not specify a version, the implementation may use any version it has available and may use any means to determine what version to use, including, but not limited to, examining the version of the stylesheet. It is *implementation-defined* which XSLT version(s) is/are supported.

The XSLT stylesheet provided on the `stylesheet` port is invoked. It is a *dynamic error* (`err:XC0093`) if a static error occurs during the static analysis of the XSLT stylesheet. Any parameters passed in the `parameters` option are used to define top-level stylesheet parameters.

Parameters passed in the `static-parameters` option are passed as static parameters to the XSLT invocation. Whether static parameters are supported is *implementation-defined* and depends on the XSLT version (which must be 3.0 or higher). If static parameters are not supported the option is ignored.

It is a *dynamic error* (`err:XC0095`) if an error occurred during the transformation. It is a *dynamic error* (`err:XC0096`) if the transformation is terminated by XSLT message termination. How XSLT message termination errors are reported to the XProc processor is *implementation-dependent*. Implementations **should** raise an error using

the error code from the XSLT step (for example, the `error-code` specified on the `xsl:message` or `Q{http://www.w3.org/2005/xqt-errors}XTTM9000` if no code is provided).

If XSLT 2.0 or XSLT 3.0 is used, the outputs of this step **may** include PSVI annotations.

The interpretation of the input and output ports as well as for the other options depends on the selected XSLT version.

### 2.48.1. Invoking an XSLT 3.0 stylesheet

The value of `global-context-item` is used as global context item for the stylesheet invocation. If no value is supplied, the following applies:

- If there is a single document on the `source` port, this document will become the value of the `global-context-item` option.
- If there are none or multiple documents on the `source` port, the global context item is absent.

The `populate-default-collection` option is used to control whether all the documents appearing on `source` port form the default collection for the XSLT transformation.

If no value is supplied for `template-name` option an "Apply-template invocation" is performed. The documents that appear on `source` are taken to be the initial match selection. if `populate-default-collection` is true, they are also the default collection. If a value is supplied for the `initial-mode` option, this value is used as the initial-mode for the invocation. It is a *dynamic error* (`err:XC0008`) if the stylesheet does not support a given mode. If no value is supplied, nothing is supplied to the invocation, so the default behaviour defined for XSLT 3.0 could be applied.

If a value is supplied for option `template-name` a "Call template invocation" is performed. The documents on port `source` are taken as the default collection in this case. Option `initial-mode` is ignored. It is a *dynamic error* (`err:XC0056`) if the stylesheet does not provide a given template.

Independent of the way the stylesheet is invoked, the principal result(s) will appear on output port `result` while secondary result(s) will appear on output port `secondary`. The order in which result documents appear on the `secondary` port is *implementation dependent*. Whether the raw results are delivered or a result tree is constructed, depends on the (explicit or implicit) setting for attribute `build-tree` of in the output-definition for the respective result. If a result tree is constructed, the result will be a text document if it is a single text node wrapped into a document node. Otherwise it will be either an XML document or an HTML document depending on the attribute `method` on the output-definition for the respective result. If no result tree is constructed, the stylesheet invocation may additionally deliver a sequence of atomic values, maps, or arrays. For each item in this sequence a JSON document will be constructed and appear on the steps output port.

Option `output-base-uri` sets the base output URI per XSLT 3.0 specification. If a final result tree is constructed, this URI is used to resolve a relative URI reference. If no value is supplied for `output-base-uri`, the base URI of the first document in the `source` port's sequence is used. If no document is supplied on port `source` the base URI of the document on port `stylesheet` is used. It is a *dynamic error* (`err:XC0121`) if a document appearing on the `secondary` port has a base URI that is not both absolute and valid according to [RFC 3986].

> **Note**
>
> If no result tree is constructed for one of secondary results, a sequence of documents sharing the same value for attribute `href` may appear on output port `result`.

#### 2.48.2. Invoking an XSLT 2.0 stylesheet

If a sequence of documents is provided on the `source` port, the first document is used as the initial context node. The whole sequence is also the default collection. If no documents are provided on the `source` port, the initial context node is undefined and the default collection is empty. It is a *dynamic error* (`err:XC0094`) if any document supplied on the source port is not an XML document, an HTML documents, or a Text document if XSLT 2.0 is used.

The `populate-default-collection` option is used to control whether all the documents appearing on `source` port form the default collection for the XSLT transformation.

The value of option `global-context-item` is ignored if a stylesheet is invoked as per XSLT 2.0. The invocation of the transformation is controlled by the `initial-mode` and `template-name` options that set the initial mode and/or named template in the XSLT transformation where processing begins. It is a *dynamic error* (`err:XC0007`) if any key in `parameters` is associated to a value which is not an instance of the XQuery 1.0 and XPath 2.0 Data Model, e.g. with a map, an array, or a function. It is a *dynamic error* (`err:XC0008`) if the specified initial mode cannot be applied to the specified stylesheet. It is a *dynamic error* (`err:XC0056`) if the specified template name cannot be applied to the specified stylesheet.

The primary result document of the transformation, if there is one, appears on the `result` port. At most one document can appear on the `result` port. All other result documents appear on the `secondary` port. The order in which result documents appear on the `secondary` port is *implementation dependent*.

The `output-base-uri` option sets the context's output base URI per the XSLT 2.0 specification, otherwise the base URI of the `result` document is the base URI of the first document in the `source` port's sequence. If no document is supplied on port `source` the base URI of the document on port `stylesheet` is used. It is a *dynamic error* (`err:XC0121`) if a document appearing on the `secondary` port has a base URI that is not both absolute and valid according to [RFC 3986].

#### 2.48.3. Invoking an XSLT 1.0 stylesheet

The document provided for `source` is used the transformations source tree. It is a *dynamic error* (`err:XC0039`) if the source port does not contain exactly one XML document or one HTML document if XSLT 1.0 is used. The values supplied for options `global-context-item`, `initial-mode`, and `template-name` are ignored. If XSLT 1.0 is used, an empty sequence of documents **must** appear on the `secondary` port. An XSLT 1.0 step **should** use the value of the `output-base-uri` as the base URI of its output, if the option is specified.

The key/value pairs supplied in `parameters` are used to set top-level parameters in the stylesheet. If the value is an atomic value or a node, its string value is supplied to the stylesheet. It is a *dynamic error* (`err:XC0105`) if an XSLT 1.0 stylesheet is invoked and option `parameters` contains a value that is not an atomic value or a node.

**Document properties**

No document properties are preserved. The `base-uri` property of each document will reflect the base URI specified by the tranformation. If the transformation does not establish a base URI, the document will not have one.

## 3. Step Errors

Several of the steps in the standard step library can generate *dynamic errors*.

A [Definition: A *dynamic error* is one which occurs while a pipeline is being evaluated.] Examples of dynamic errors include references to URIs that cannot be resolved, steps which fail, and pipelines that exhaust the capacity of an implementation (such as memory or disk space).

If a step fails due to a dynamic error, failure propagates upwards until either a `p:try` is encountered or the entire pipeline fails. In other words, outside of a `p:try`, step failure causes the entire pipeline to fail.

Dynamic errors raised by steps are divided into two categories: dynamic errors and step errors. The distinction is largely that "step errors" tend to be related to a particular step or small group of steps (e.g., validation error) whereas the "dynamic errors" apply to many more steps (e.g., URI not available). There is also precedent for some of the error codes dating back to XProc 1.0.

*Dynamic Errors*

`err:XD0011`
: It is a dynamic error if the resource referenced by the href option does not exist, cannot be accessed or is not a file.

See: Handling of ZIP archives, p:load

`err:XD0023`
It is a dynamic error if a DTD validation is performed and either the document is not valid or no DTD is found.

See: Loading XML data

`err:XD0043`
It is a dynamic error if the dtd-validate parameter is true and the processor does not support DTD validation.

See: Loading XML data

`err:XD0049`
It is a dynamic error if the text value is not a well-formed XML document

See: Casting from a text media type, Loading XML data

`err:XD0057`
It is a dynamic error if the text document does not conform to the JSON grammar, unless the parameter liberal is true and the processor chooses to accept the deviation.

See: Casting from a text media type, Loading JSON data

`err:XD0058`
It is a dynamic error if the parameter duplicates is reject and the text document contains a JSON object with duplicate keys.

See: Casting from a text media type, Loading JSON data

`err:XD0059`
It is a dynamic error if the parameter map contains an entry whose key is defined in the specification of fn:parse-json and whose value is not valid for that key, or if it contains an entry with the key fallback when the parameter escape with true() is also present.

See: Casting from a text media type, Loading JSON data

`err:XD0060`
It is a dynamic error if the text document can not be converted into the XPath data model

See: Casting from a text media type, Loading text data

`err:XD0062`
It is a dynamic error if the content-type is specified and the document-properties has a “content-type” that is not the same.

See: p:load

`err:XD0064`
It is a dynamic error if the base URI is not both absolute and valid according to .

See: p:archive, p:archive-manifest, p:load, p:make-absolute-uris, p:set-properties, p:unarchive

`err:XD0070`
It is a dynamic error if a value is assigned to the serialization document property that cannot be converted into map(xs:QName, item()*) according to the rules in section “QName handling” of .

See: p:set-properties

`err:XD0078`
It is a dynamic error if the loaded document cannot be represented as an HTML document in the XPath data model.

See: Loading HTML data

`err:XD0079`
It is a dynamic error if a supplied content-type is not a valid media type of the form “type/subtype+ext” or “type/subtype”.

See: Overriding content types, p:cast-content-type, p:http-request, p:http-request, p:load, p:text-join, p:uncompress

*Step Errors*

`err:XC0001`
It is a dynamic error if the value of option override-content-type is not a text media type.

See: p:text-join

`err:XC0003`
: It is a dynamic error if a “username” or a “password” key is present without specifying a value for the “auth-method” key, if the requested auth-method isn't supported, or the authentication challenge contains an authentication method that isn't supported.

  See: p:http-request

`err:XC0007`
: It is a dynamic error if any key in parameters is associated to a value which is not an instance of the XQuery 1.0 and XPath 2.0 Data Model, e.g. with a map, an array, or a function.

  See: Invoking an XSLT 2.0 stylesheet

`err:XC0008`
: It is a dynamic error if the stylesheet does not support a given mode.

  See: Invoking an XSLT 3.0 stylesheet, Invoking an XSLT 2.0 stylesheet

`err:XC0009`
: It is a dynamic error if the specified XQuery version is not available.

  See: p:xquery

`err:XC0013`
: It is a dynamic error if the pattern matches a processing instruction and the new name has a non-null namespace.

  See: p:rename

`err:XC0014`
: It is a dynamic error if the XML namespace (http://www.w3.org/XML/1998/namespace) or the XMLNS namespace (http://www.w3.org/2000/xmlns/) is the value of either the from option or the to option.

  See: p:namespace-rename

`err:XC0019`
: It is a dynamic error if the documents are not equal according to the specified comparison method, and the value of the fail-if-not-equal option is true.

  See: p:compare

`err:XC0023`
: It is a dynamic error if the selection pattern matches a node which is not an element.

  See: p:add-attribute, p:delete, p:insert, p:label-elements, p:make-absolute-uris, p:rename, p:replace, p:set-attributes, p:unwrap, p:wrap

`err:XC0024`
: It is a dynamic error if the selection pattern matches a document node and the value of the position is “before” or “after”.

  See: p:insert

`err:XC0025`
: It is a dynamic error if the selection pattern matches anything other than an element or a document node and the value of the position option is “first-child” or “last-child”.

  See: p:insert

`err:XC0029`
: It is a dynamic error if an XInclude error occurs during processing.

  See: p:xinclude

`err:XC0030`
: It is a dynamic error if the response body cannot be interpreted as requested (e.g. application/json to override application/xml content).

  See: p:http-request

`err:XC0036`
: It is a dynamic error if the requested hash algorithm is not one that the processor understands or if the value or parameters are not appropriate for that algorithm.

  See: p:hash

`err:XC0037`
: It is a dynamic error if the value provided is not a properly x-www-form-urlencoded value.

  See: p:www-form-urldecode

`err:XC0038`
: It is a dynamic error if the specified xslt version is not available.

  See: p:xslt

`err:XC0039`
: It is a dynamic error if the source port does not contain exactly one XML document or one HTML document if XSLT 1.0 is used.

  See: Invoking an XSLT 1.0 stylesheet

`err:XC0050`
: It is a dynamic error if the URI scheme is not supported or the step cannot store to the specified location.

  See: p:store

`err:XC0056`
: It is a dynamic error if the stylesheet does not provide a given template.

  See: Invoking an XSLT 3.0 stylesheet, Invoking an XSLT 2.0 stylesheet

`err:XC0058`
: It is a dynamic error if the all and relative options are both true.

  See: p:add-xml-base

`err:XC0059`
: It is a dynamic error if the QName value in the attribute-name option uses the prefix “xmlns” or any other prefix that resolves to the namespace name http://www.w3.org/2000/xmlns/.

  See: p:add-attribute

`err:XC0060`
: It is a dynamic error if the processor does not support the specified version of the UUID algorithm.

  See: p:uuid

`err:XC0062`
: It is a dynamic error if the match option matches a namespace node.

See: p:delete

`err:XC0069`
It is a dynamic error if the properties map contains a key equal to the string "content-type".

See: p:set-properties

`err:XC0071`
It is a dynamic error if the p:cast-content-type step cannot perform the requested cast.

See: p:cast-content-type

`err:XC0072`
It is a dynamic error if the c:data contains content is not a valid base64 string.

See: Casting from an XML media type

`err:XC0073`
It is a dynamic error if the c:data element does not have a content-type attribute.

See: Casting from an XML media type

`err:XC0074`
It is a dynamic error if the content-type is supplied and is not the same as the content-type specified on the c:data element.

See: Casting from an XML media type

`err:XC0076`
It is a dynamic error if the comparison method specified in p:compare is not supported by the implementation.

See: p:compare

`err:XC0077`
It is a dynamic error if the media types of the documents supplied are incompatible with the comparison method.

See: p:compare

`err:XC0078`
: It is a dynamic error if the value associated with the “fail-on-timeout” is associated with true() and a HTTP status code 408 is encountered.

  See: p:http-request

`err:XC0079`
: It is a dynamic error if the map parameters contains an entry whose key is defined by the implementation and whose value is not valid for that key.

  See: p:archive, p:archive-manifest, p:cast-content-type, p:compress, p:unarchive, p:uncompress

`err:XC0080`
: It is a dynamic error if the number of documents on the archive does not match the expected number of archive input documents for the given format and command.

  See: Handling of ZIP archives

`err:XC0081`
: It is a dynamic error if the format of the archive does not match the format as specified in the format option.

  See: p:archive

`err:XC0084`
: It is a dynamic error if two or more documents appear on the p:archive step's source port that have the same base URI or if any document that appears on the source port has no base URI.

  See: p:archive

`err:XC0085`
: It is a dynamic error if the format of the archive does not match the specified format, cannot be understood, determined and/or processed.

  See: p:archive, p:archive-manifest, p:unarchive

`err:XC0092`
: It is a dynamic error if as a consequence of changing or removing the namespace of an attribute the attribute's name is not unique on the respective element.

See: p:namespace-rename

`err:XC0093`
It is a dynamic error if a static error occurs during the static analysis of the XSLT stylesheet.

See: p:xslt

`err:XC0094`
It is a dynamic error if any document supplied on the source port is not an XML document, an HTML documents, or a Text document if XSLT 2.0 is used.

See: Invoking an XSLT 2.0 stylesheet

`err:XC0095`
It is a dynamic error if an error occurred during the transformation.

See: p:xslt

`err:XC0096`
It is a dynamic error if the transformation is terminated by XSLT message termination.

See: p:xslt

`err:XC0098`
It is a dynamic error if a dynamic XPath error occurred while applying sort-key to a line.

See: p:text-sort

`err:XC0099`
It is a dynamic error if the result of applying sort-key to a given line results in a sequence with more than one item.

See: p:text-sort

`err:XC0100`
It is a dynamic error if the document on port manifest does not conform to the given schema.

See: p:archive

`err:XC0101`
: It is a dynamic error if a document appearing on port source cannot be represented in the XDM version associated with the chosen XQuery version, e.g. when a JSON document contains a map and XDM 3.0 is used.

  See: p:xquery

`err:XC0102`
: It is a dynamic error if any key in option parameters is associated to a value that cannot be represented in the XDM version associated with the chosen XQuery version, e.g. with a map, an array, or a function when XDM 3.0 is used.

  See: p:xquery

`err:XC0103`
: It is a dynamic error if any error occurs during XQuery's static analysis phase.

  See: p:xquery

`err:XC0104`
: It is a dynamic error if any error occurs during XQuery's dynamic evaluation phase.

  See: p:xquery

`err:XC0105`
: It is a dynamic error if an XSLT 1.0 stylesheet is invoked and option parameters contains a value that is not an atomic value or a node.

  See: Invoking an XSLT 1.0 stylesheet

`err:XC0106`
: It is a dynamic error if duplicate keys are encountered and option duplicates has value "reject".

  See: p:json-merge

`err:XC0107`
: It is a dynamic error if a document of a not supported document type appears on port source of p:json-merge.

  See: p:json-merge

`err:XC0108`
: It is a dynamic error if any prefix is not in-scope at the point where the p:namespace-delete occurs.

  See: p:namespace-delete

`err:XC0109`
: It is a dynamic error if a namespace is to be removed from an attribute and the element already has an attribute with the resulting name.

  See: p:namespace-delete

`err:XC0110`
: It is a dynamic error if the evaluation of the XPath expression in option key for a given item returns either a sequence, an array, a map, or a function.

  See: p:json-merge

`err:XC0111`
: It is a dynamic error if a document of an unsupported document type appears on port source of p:json-join.

  See: p:json-join

`err:XC0112`
: It is a dynamic error if more than one document appears on the port manifest.

  See: p:archive

`err:XC0118`
: It is a dynamic error if an archive manifest is invalid according to the specification.

  See: The archive manifest

`err:XC0119`
: It is a dynamic error if flatten is neither “unbounded”, nor a string that may be cast to a non-negative integer.

  See: p:json-join

`err:XC0120`
: It is a dynamic error if the relative-to option is not present and the document on the source port does not have a base URI.

  See: p:archive-manifest, p:unarchive

`err:XC0121`
: It is a dynamic error if a document appearing on the secondary port has a base URI that is not both absolute and valid according to .

  See: Invoking an XSLT 3.0 stylesheet, Invoking an XSLT 2.0 stylesheet

`err:XC0122`
: It is a dynamic error if the given method is not supported.

  See: p:http-request

`err:XC0123`
: It is a dynamic error if any key in the “auth” map is associated with a value that is not an instance of the required type.

  See: p:http-request

`err:XC0124`
: It is a dynamic error if any key in the “parameters” map is associated with a value that is not an instance of the required type.

  See: p:http-request

`err:XC0125`
: It is a dynamic error if the key “accept-multipart” as the value false() and a multipart response is detected.

  See: p:http-request

`err:XC0126`
: It is a dynamic error if the XPath expression in assert evaluates to false.

  See: p:http-request

`err:XC0127`
: It is a dynamic error if the headers map contains two keys that are the same when compared in a case-insensitive manner.

See: p:http-request

`err:XC0128`
It is a dynamic error if the URI's scheme is unknown or not supported.

See: p:http-request

`err:XC0131`
It is a dynamic error if the processor cannot support the requested encoding.

See: p:http-request

`err:XC0132`
It is a dynamic error if the override content encoding cannot be supported.

See: p:http-request

`err:XC0133`
It is a dynamic error if more than one document appears on the source port and a content-type header is present and the content type specified is not a multipart content type.

See: Construction of a multipart request

`err:XC0146`
It is a dynamic error if the specified value for the override-content-types option is not an array of arrays, where the inner arrays have exactly two members of type xs:string.

See: Overriding content types

`err:XC0147`
It is a dynamic error if the specified value is not a valid XPath regular expression.

See: Overriding content types, p:text-replace, p:unarchive

`err:XC0150`
It is a dynamic error if evaluating the XPath expression in option test on a context document results in an error.

See: p:split-sequence

`err:XC0201`
: It is a dynamic error if the p:uncompress step cannot perform the requested content-type cast.

  See: p:uncompress

`err:XC0202`
: It is a dynamic error if the compression format cannot be understood, determined and/or processed.

  See: p:compress, p:uncompress

`err:XC0203`
: It is a dynamic error if the specified boundary is not valid (for example, if it begins with two hyphens "--").

  See: Construction of a multipart request

# A. Conformance

Conformant processors **must** implement all of the features described in this specification except those that are explicitly identified as optional.

Some aspects of processor behavior are not completely specified; those features are either *implementation-dependent* or *implementation-defined*.

[Definition: An *implementation-dependent* feature is one where the implementation has discretion in how it is performed. Implementations are not required to document or explain how *implementation-dependent* features are performed.]

[Definition: An *implementation-defined* feature is one where the implementation has discretion in how it is performed. Conformant implementations **must** document how *implementation-defined* features are performed.]

## A.1. Implementation-defined features

The following features are implementation-defined:

1. The list of formats supported by the p:archive step is implementation-defined. See Section 2.3, “p:archive”.
2. The list of archive formats that can be modified by p:archive is implementation-defined. See Section 2.3, “p:archive”.
3. The semantics of any additional attributes, elements and their values are implementation-defined. See Section 2.3, “p:archive”.
4. It is implementation-defined what other formats are supported. See Section 2.3, “p:archive”.
5. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.3, “p:archive”.
6. It is implementation-defined what other formats are supported. See Section 2.3, “p:archive”.
7. It is implementation-defined how the step determines the archive's format. See Section 2.3, “p:archive”.
8. The c:archive root element may contain additional implementation-defined attributes. See Section 2.3.1, “The archive manifest”.
9. The default compression method is implementation-defined. See Section 2.3.1, “The archive manifest”.
10. It is implementation-defined what other compression methods are supported. See Section 2.3.1, “The archive manifest”.
11. The default compression method is implementation-defined. See Section 2.3.1, “The archive manifest”.
12. It is implementation-defined what compression levels are supported. See Section 2.3.1, “The archive manifest”.
13. The c:entry elements may contain additional implementation-defined attributes. See Section 2.3.1, “The archive manifest”.
14. The p:archive step may support additional, implementation-defined commands for ZIP files. See Section 2.3.2, “Handling of ZIP archives”.
15. The actual parameter names supported by p:archive for a particular format are implementation-defined. See Section 2.3.2, “Handling of ZIP archives”.
16. It is implementation-defined what other formats are supported. See Section 2.4, “p:archive-manifest”.

17. It is implementation-defined how the step determines the archive's format. See Section 2.4, “p:archive-manifest”.
18. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.4, “p:archive-manifest”.
19. Additional information provided for entries in p:archive-manifest is implementation-defined. See Section 2.4, “p:archive-manifest”.
20. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.5, “p:cast-content-type”.
21. The precise nature of the conversion from XML to JSON is implementation-defined. See Section 2.5.1, “Casting from an XML media type”.
22. Casting from an XML media type to any other media type when the input document is not a c:data document is implementation-defined. See Section 2.5.1, “Casting from an XML media type”.
23. Casting from an HTML media type to a JSON media type is implementation-defined. See Section 2.5.2, “Casting from an HTML media type”.
24. Casting from an HTML media type to any other media type is implementation-defined. See Section 2.5.2, “Casting from an HTML media type”.
25. It is implementation-defined whether other result formats are supported. See Section 2.5.3, “Casting from a JSON media type”.
26. Casting from a JSON media type to an HTML media type is implementation-defined. See Section 2.5.3, “Casting from a JSON media type”.
27. Casting from a JSON media type to any other media type is implementation-defined. See Section 2.5.3, “Casting from a JSON media type”.
28. The precise way in which text documents are parsed into the XPath data model is implementation-defined. See Section 2.5.4, “Casting from a text media type”.
29. Casting from a text media type to any other media type is implementation-defined. See Section 2.5.4, “Casting from a text media type”.
30. Casting from any other media type to a HTML media type, a JSON media type or a text document is implementation-defined. See Section 2.5.5, “Casting from any other media type”.
31. Casting from any other media type to any other media type is implementation-defined. See Section 2.5.5, “Casting from any other media type”.

32. Implementations of p:compare must support the deep-equal method; other supported methods are implementation-defined. See Section 2.6, “p:compare”.
33. If fail-if-not-equal is false, and the documents differ, an implementation-defined summary of the differences between the two documents may appear on the differences port. See Section 2.6, “p:compare”.
34. It is implementation-defined what other formats are supported. See Section 2.7, “p:compress”.
35. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.7, “p:compress”.
36. It is implementation-defined what other algorithms are supported. See Section 2.12, “p:hash”.
37. It is implementation-defined how a multipart boundary is constructed. See Section 2.13.1, “Construction of a multipart request”.
38. In the absence of an explicit type, the content type is implementation-defined See Section 2.19, “p:load”.
39. Additional XML parameters are implementation-defined. See Section 2.19.1, “Loading XML data”.
40. Text parameters are implementation-defined. See Section 2.19.2, “Loading text data”.
41. Additional JSON parameters are implementation-defined. See Section 2.19.3, “Loading JSON data”.
42. The precise way in which HTML documents are parsed into the XPath data model is implementation-defined. See Section 2.19.4, “Loading HTML data”.
43. HTML parameters are implementation-defined. See Section 2.19.4, “Loading HTML data”.
44. How a processor interprets other media types is implementation-defined. See Section 2.19.5, “Loading binary data”.
45. Parameters for other media types are implementation-defined. See Section 2.19.5, “Loading binary data”.
46. Support for other collations is implementation-defined. See Section 2.36, “p:text-sort”.

47. It is implementation-defined what other formats are supported. See Section 2.38, “p:unarchive”.
48. It is implementation-defined how the step determines the archive's format. See Section 2.38, “p:unarchive”.
49. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.38, “p:unarchive”.
50. It is implementation-defined what other formats are supported. See Section 2.39, “p:uncompress”.
51. It is implementation-defined how the step determines the compression format. See Section 2.39, “p:uncompress”.
52. The semantics of the keys and the allowed values for these keys are implementation-defined. See Section 2.39, “p:uncompress”.
53. If the version is not specified, the version of UUID computed is implementation-defined. See Section 2.41, “p:uuid”.
54. Support for other versions of UUID, and the mechanism by which the necessary inputs are made available for computing other versions, is implementation-defined. See Section 2.41, “p:uuid”.
55. Support for XQueryX is implementation-defined. See Section 2.47, “p:xquery”.
56. The point in time returned as the current dateTime is implementation-defined. See Section 2.47, “p:xquery”.
57. The implicit timezone is implementation-defined. See Section 2.47, “p:xquery”.
58. It is implementation-defined which XSLT version(s) is/are supported. See Section 2.48, “p:xslt”.
59. Whether static parameters are supported is implementation-defined and depends on the XSLT version (which must be 3.0 or higher). See Section 2.48, “p:xslt”.

## A.2. Implementation-dependent features

The following features are implementation-dependent:

1. If the IRI reference specified by the base-uri option on p:make-absolute-uris is absent and the input document has no base URI, the results are implementation-dependent. See Section 2.20, “p:make-absolute-uris”.
2. The set of available documents (those that may be retrieved with a URI) is implementation-dependent. See Section 2.47, “p:xquery”.
3. The set of available collections is implementation-dependent. See Section 2.47, “p:xquery”.
4. How XSLT message termination errors are reported to the XProc processor is implementation-dependent. See Section 2.48, “p:xslt”.

# B. References

## B.1. Normative References

[XProc 3.0] *XProc 3.0: An XML Pipeline Language*. Norman Walsh, Achim Berndzen, Gerrit Imsieke and Erik Siegel, editors.

[W3C XML Schema: Part 2] *XML Schema Part 2: Datatypes Second Edition*. Paul V. Biron and Ashok Malhotra, editors. World Wide Web Consortium, 28 October 2004.

[XPath 3.1] *XML Path Language (XPath) 3.1*. Jonathan Robie, Michael Dyck, Josh Spiegel, editors. W3C Recommendation. 21 March 2017.

[XPath and XQuery Functions and Operators 3.1] *XPath and XQuery Functions and Operators 3.1*. Michael Kay, editor. W3C Recommendation. 21 March 2017

[XSLT 3.0] *XSL Transformations (XSLT) Version 3.0*. Michael Kay, editor. W3C Recommendation. 8 June 2017.

[XInclude] *XML Inclusions (XInclude) Version 1.0 (Second Edition)*. Jonathan Marsh, David Orchard, and Daniel Veillard, editors. W3C Recommendation. 15 November 2006.

[RFC 1321] *RFC 1321: The MD5 Message-Digest Algorithm*. R. Rivest. Network Working Group, IETF, April 1992.

[RFC 1521] *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. N. Borenstein. Network Working Group, IETF, September 1993.

[RFC 2046] *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. N. Freed, N. Borenstein, editors. Internet Engineering Task Force. November, 1996.

[RFC 2119] *Key words for use in RFCs to Indicate Requirement Levels*. S. Bradner. Network Working Group, IETF, Mar 1997.

[RFC 2617] *RFC 2617: HTTP Authentication: Basic and Digest Access Authentication*. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart. June, 1999 .

[RFC 3986] *RFC 3986: Uniform Resource Identifier (URI): General Syntax*. T. Berners-Lee, R. Fielding, and L. Masinter, editors. Internet Engineering Task Force. January, 2005.

[UUID] *ITU X.667: Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components*. 2004.

[SHA1] *Federal Information Processing Standards Publication 180-1: Secure Hash Standard*. 1995.

[CRC32] "32-Bit Cyclic Redundancy Codes for Internet Applications", *The International Conference on Dependable Systems and Networks: 459*. 10.1109/DSN.2002.1028931. P. Koopman. June 2002.

[ZIP] *.ZIP File Format Specification*.

[GZIP] *GZIP file format specification version 4.3*.

# C. Glossary

***dynamic error***

A *dynamic error* is one which occurs while a pipeline is being evaluated.

***implementation-defined***
: An *implementation-defined* feature is one where the implementation has discretion in how it is performed. Conformant implementations **must** document how *implementation-defined* features are performed.

***implementation-dependent***
: An *implementation-dependent* feature is one where the implementation has discretion in how it is performed. Implementations are not required to document or explain how *implementation-dependent* features are performed.

# D. Ancillary files

This specification includes by reference a number of ancillary files.

**steps.xpl**
: An XProc step library for the declared steps.

# E. Credits

This document is derived from XProc: An XML Pipeline Language published by the W3C. It was developed by the *XML Processing Model Working Group* and edited by Norman Walsh, Alex Miłowski, and Henry Thompson.

The editors of this specification extend their gratitude to everyone who contributed to this document and all of the versions that came before it.

# F. Change Log

This appendix catalogs non-editorial changes made after the August 2020 "last call" draft:

1. Clarified that the manifest has precedence in the `p:archive` step. (issue 462.)
2. Changed the type of several options from `xs:token` to `xs:string` (issue 490.)

3. Changed the type of the `parameters` option from `map(xs:string,xs:untypedAtomic+)` to `map(xs:string,xs:anyAtomicType+)`. (issue 491.)

These are the non-editorial changes made after the February 2020 "last call" draft:

1. For `p:cast-content-type` the expected result type for casting a `c:param-set` document to "`application/json`" was specified as `map(xs:QName, xs:string)`. (2020-03-15)

2. In `p:http-request`, instead of using all document properties (with a few explicit exceptions) as headers, only document properties in the `http://www.w3.org/ns/xproc-http` namespace will be used. (2020-03-18)

3. Section 2.3.1, "The archive manifest": An attribute `c:entry/@content-type` was added to the archive manifest, to be filled by the `p:archive-manifest` step. (2020-03-20)

4. A `static-parameters` was added to `p:xslt`. (2020-03-23)

5. The `override-content-types` option was added to `p:archive-manifest` and `p:unarchive`. (2020-03-30)

6. Clarified the regular expression matching for `p:text-replace` and `p:unarchive`. Added an error code for invalid regular expressions. (2020-04-02)

7. Replaced errors XC0070 and XC0130 with XD0079. (2020-04-09)

8. Changed signature of `p:split-sequence` so that any document can appear one port source. (2020-05-22)

9. Change the behavior of the `global-context-item` option of `p:xslt`. (2020-06-10)

10. Clarified which steps may produce PSVI annotations. (2020-06-09)

11. Clarified the behaviour in `p:archive`: A missing resource referenced by c:archive/c:entry/@href is only an error for command = 'create'. (2020-06-11)

12. Option `populate-default-collection` is added to the signature of `p:xslt`. (2020-06-20)

13. Clarified how the default `content-type` header of `p:http-request` is constructed if a single document appears on `source` port. (2020-06-20)

14. Added error (XD0079) to `p:http-request` and `p:load` for invalid content-types. (2020-06-23)

15. Changed signature of the `result` port of `p:load` to `sequence="false"` and adapted the prose accordingly. (2020-06-24)